

**1966  
to  
1985**

# **ACM Turing Award Lectures The First Twenty Years**

ACM PRESS ANTHOLOGY SERIES



ACM Press  
New York, New York



**Addison-Wesley Publishing Company**

Reading Massachusetts • Menlo Park California  
Don Mills Ontario • Wokingham England • Amsterdam  
Sydney • Singapore • Tokyo • Madrid  
Bogota • Santiago • San Juan

# Лекции лауреатов премии Тьюринга

за первые  
двадцать лет  
1966–1985

Перевод с английского  
под редакцией Ю. М. Баяковского



МОСКВА «МИР»

1993



ББК 32.97  
Л43  
УДК 519.6

Авторы: Перлис А. Дж., Дейкстра Э. В., Кнут Д. Е., Скотт Д. С., Бэкус Дж., Флойд Р., Хоар Ч. Э. Р., Ритчи Д. М., Томпсон К., Вирт Н., Уилкс М. В., Хэмминг Р. В., Минский М., Уилкинсон Дж., Маккарти Дж., Бахман Ч. В., Ньюэлл А., Саймон Х., Рабин М. О., Айверсон К. Е., Кодд Э. Д., Кук С. А., Карп Р. М., Френкель К.

Л43      **Лекции лауреатов премии Тьюринга:** Пер. с англ./Под ред. Р. Эшенхёрста. — М.: Мир, 1993. — 560 с., ил.  
ISBN 5—03—002130—2

Книга включает лекции лауреатов премии Тьюринга, которой отмечаются выдающиеся достижения в области программирования, вычислительной математики, математического обеспечения ЭВМ. Среди лауреатов — известные специалисты: Э. Дейкстра (Нидерланды), Д. Кнут (США), Ч. Хоар (Великобритания), Н. Вирт (Швейцария) и др., книги которых переводились на русский язык и известны советским читателям. Ряд лекций сопровождается послесловиями авторов, отражающими современное состояние и перспективы исследований.

Для математиков-прикладников, программистов разной квалификации, аспирантов и студентов вузов.

Л 2404010000—052      КБ—46—37  
041(01)—93

ББК 32.97'

*Редакция литературы по математическим наукам*

ISBN 5—03—002130—2 (русск.)  
ISBN 0—201—07794—9 (англ.)

© 1987 by the ACM Press, A. Division of the Association for Computing Machinery, Inc. (ACM).

© перевод на русский язык, коллектив переводчиков, 1993.

## ОТ РЕДАКТОРА ПЕРЕВОДА

В Ваших руках, читатель, не совсем обычная, а скорее всего совсем необычная книга. У нее коллективный автор — 23 крупнейших ученых в области информатики, их имена хорошо знакомы специалистам. Книга писалась 20 лет, и с каждой из этих двадцати ступенек она позволяет бросить ретроспективный взгляд на пройденный путь, ощутить напряжение современности, заглянуть в недалекое будущее. Способность подняться до философского осмысления науки и является той главной характерной чертой, которая объединила Тьюринговских лауреатов в это созвездие.

Я поступил бы наивно, если бы попытался проанализировать содержащуюся в книге коллекцию эссе и высказать свои рекомендации. Думаю каждый найдет в книге что-то для себя, с чем-то безусловно согласится, а с чем-то, возможно, поспорит. Однако на некоторые вопросы (а они несомненно возникнут) все-таки стоит ответить. В частности, что за организация АСМ, чем она известна, кроме Тьюринговских премий, какую роль она играет в обществе, есть ли что-нибудь подобное в привычной для нас действительности и если нет, то почему?

Association for Computing Machinery (буквально Ассоциация по вычислительным механизмам) была основана в США в 1947 г., через год после появления первой электронной вычислительной машины ЭНИАК. В самом названии Ассоциации звучит что-то старинное и забытое. (Впрочем: и мои сверстники, выпускники 1960—1965 гг. получили образование по специальности «Счетно-решающие приборы и устройства».) Высказывались неоднократно предложения осовременить название, но приверженность истории и традициям охраняет его. Утверждение высоких профессиональных стандартов и традиций, повышение престижа профессии, содействие прогрессу науки и применениям новой технологии — таковы цели АСМ.

Основанная по инициативе 7 специалистов и объединившая поначалу 200 человек, АСМ в настоящее время насчитывает около 80 000 членов. С развитием и структурированием информатики происходило структурирование Ассоциации, в которой теперь 33 специальные группы по интересам (SIG). Наиболее

крупные из них SIGGRAPH (машинная графика), SIGSOFT (программная инженерия), SIGPLAN (языки программирования), SIGART (искусственный интеллект). В 1990 г. ACM и ее структурные подразделения провели более 75 конференций с числом участников превысившим 50 000. Ассоциация издает 12 журналов; кроме того, каждая группа (SIG) имеет свой бюллетень. Бюджет организации составляет около 25 млн. долларов, складывающийся из членских взносов, поступлений от издательской деятельности, конференций, рекламы.

Ничего подобного ни по масштабам, ни по характеру деятельности в нашей стране нет, хотя еще на Первой Всесоюзной конференции по программированию (ВКП-1, Киев, 1968 г.) предложения на эту тему обсуждались. Так в чем же причина? Дело, разумеется, в том, что иерархическая система, которую называют еще и тоталитарной, с великим подозрением, а следовательно, и нетерпимо относится к любым самостоятельным организациям, стремясь обязательно включить их в структуру ведомственной подчиненности. Характерный факт: в Уставе ВОИВТ (Всесоюзное общество по информатике и вычислительной технике), образованного уже в пору «перестройки», декларируется, что оно работает под руководством КПСС и исповедует в качестве организационного принципа — демократический централизм. В России профессиональные ассоциации, подобные ACM, еще предстоит создать, и опыт ACM в немалой степени может этому помочь.

Еще один момент, заслуживающий внимания, — это собственно премии Тьюринга. Из 9 премий, которыми ACM отмечает специалистов за их научные достижения и особые профессиональные заслуги, премия Тьюринга наиболее почетна и престижна. Ежегодно она присуждается только одному человеку (исключения составляют те случаи, когда трудно выделить вклад каждого из участников совместной работы). Премия присуждается за совершенно конкретные результаты и в этом можно убедиться, прочитав представления лауреатов. Лауреатом нельзя стать вторично. Если говорить о нашей узкопрофессиональной области, то премия Тьюринга в чем-то сродни Нобелевской премии.

В нашей стране используются иные процедуры формирования научной элиты, принципиально отличные от принятых в ACM. И это необходимо иметь в виду. Можно выделить два практиковавшихся у нас варианта: избрание в члены Академии и присуждение Ленинской или Государственной премий. Об изъянах и несовершенствах каждого варианта написано много, и здесь нет необходимости останавливаться на этом.

Стоит ли удивляться, что «научная и техническая общественность» плохо знает не только выдающиеся достижения сво-

их корифеев, но и их имена. Нам многое еще предстоит заново осмысливать и переосмысливать. Поэтому важным мне представляется не только научное, историческое, но и социальное значение Тьюринговских лекций.

Идея издания лекций Тьюринговских лауреатов родилась в издательстве «Мир» лет 10 назад. Но в ту пору сами лауреаты не позволили осуществить эту идею, выразив таким образом свое отношение к нарушению прав человека в СССР. Многое изменилось с тех пор. И АСМ уже помогает преодолевать последствия нашей изоляции, оказывая содействие в проведении конференций, помогая формированию самостоятельных профессиональных структур. Предисловие к этой книге, написанное президентом АСМ Джоном Уайтом, еще одно тому подтверждение. Пользуясь случаем, я хочу выразить ему и другим нашим коллегам из АСМ признательность за сотрудничество и помощь.

Очень надеюсь, что русское издание Лекций будет содействовать интеграции наших ученых и инженеров в мировое профессиональное сообщество. И может быть, придет время, когда имя читателя этой книги появится в почетном ряду Тьюринговских лауреатов. Дерзайте!

*Ю. Баяковский  
Москва  
январь 1992 г.*

## ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Премия А. М. Тьюринга — самая первая и самая престижная премия Ассоциации информатики и вычислительной техники (The Association for Computing Machinery — ACM). Эта премия присуждается ежегодно с 1966 г. и ею отмечаются личные заслуги тех специалистов, которые внесли существенный вклад, имеющий непреходящее значение для информатики. Премия Тьюринга по сути своей международная, и многие ее лауреаты живут за пределами Соединенных Штатов; она считается самой почетной премией в информатике.

Я с удовлетворением воспринимаю издание Тьюринговских лекций на русском языке. Это событие соотносится с «открытием миру» новых государств, прежде составлявших Советский Союз, и с желанием говорящих по-русски специалистов в области информатики стать активными членами мирового научного сообщества.

Ассоциация АСМ в полной мере и активно поддерживает усилия внутри прежних советских республик и особенно в России, направленные на преодоление научно-технической изоляции времен холодной войны, и стремление включиться в глобальную деятельность по развитию вычислительных и информационных технологий. При активной поддержке Ассоциации АСМ и ее Групп по специальным интересам, в России образован ряд локальных групп АСМ. Более того, в Российской республике все чаще проходят международные семинары и конференции по информатике.

Цель ассоциации АСМ и состоит в том, чтобы способствовать развитию поистине международного вычислительного сообщества. Перед всеми нами стоят трудные проблемы: обеспечить в следующем столетии приемлемое, а это значит — высокое качество жизни. Решение этих проблем потребует совершенных вычислительных информационных технологий и повсеместного применения этих технологий для преодоления трудностей, с которыми мы сталкиваемся сегодня.

*Джон Р. Уайт  
Президент АСМ  
декабрь 1991  
Пало Альто, Калифорния,  
США.*

## ПРЕДИСЛОВИЕ

В настоящей книге, являющейся первым томом в серии антологий, намеченных к выпуску издательством ACM Press, собраны 22 очерка, представляющих Тьюринговские лекции ACM за первые 20 лет (1966—1985 гг.) существования Тьюринговских премий. Эта премия присуждается ежегодно «исследователю, выбранному за полезные сообществу программистов и пользователей достижения технического характера», которые сочтены имеющими важное и непреходящее значения для прогресса информатики. Премия учреждается в честь Алана М. Тьюринга, английского математика, работы которого «захватили воображение и вдохновили деятельность целого поколения ученых», как сказал Алан Дж. Перлис, первый лауреат этой премии.

Ежегодно лауреат Тьюринговской премии выступает с лекцией на осенней конференции ACM и получает приглашение подготовить к публикации печатный вариант своего выступления. С 1966 по 1970 г. такие работы публиковались в *Journal of the ACM*; с 1972 г. они появляются в *Communications of the ACM*. В 1975, 1976 и 1983 гг. премия присуждалась за совместную работу сразу двум лицам, и в этих случаях лауреаты либо подготавливали совместный доклад (в 1975 г.), либо каждый из них читал самостоятельную лекцию. Всего было 23 лауреата премии и 22 лекции (из которых 21 была опубликована за 20-летний период).

Первоначально планировалось представить эти лекции в хронологическом порядке, чтобы читатель мог получить общее представление о времени, к которому относится каждая лекция, из ее расположения в настоящей книге. Другая возможность состояла в тематическом принципе организации материала. Окончательное решение представляет собой компромисс между этими двумя подходами. Поскольку 10 лекций из 22 связаны с важными темами, относящимися к общей рубрике «Системы и языки программирования», было решено поместить их в хронологическом порядке в качестве части I настоящей антологии; остальные 12 лекций, удачно охватывающие весь спектр направлений в рамках общей рубрики «Компьютеры и мето-

логии программирования», составили часть II. Эта дихотомия основана на классификационной схеме, принятой в журнале ACM Computing Reviews, и в конце книги помещен указатель, показывающий место каждой из 23 публикаций в этой основной классификационной схеме информатики.

Каждая из лекций, первоначально опубликованных в Communications, сопровождалась некоторыми вводными материалами, которые перепечатываются здесь. Кроме того, каждому лауреату предоставлялась возможность высказать свои замечания, оценив с сегодняшних позиций утверждения и прогнозы, сделанные в их лекциях многолетней давности. Эти замечания включены в антологию в виде «постскриптов» и помещены после соответствующих лекций. Поскольку текст лекции 1971 г. никогда не публиковался, мы попросили Джона Маккарти написать более обширный постскриптум, рассказывающий и об этой лекции, и о его современных взглядах. Постскриптумы к лекции Ричарда Карпа 1985 г. состоят из короткого эссе «Сборка теории сложности из кусков» и интервью с Р. Карпом; оба они написаны обозревателем журнала Communications Карен Френкель и помещены после его лекции.

Сузан Л. Грэхем, являющаяся главным редактором журнала ACM Transactions on Programming Languages and Systems со времени его основания в 1978 г., написала введение к части I, поместив в соответствующий контекст включенные в эту часть работы. Я написал введение к части II, попытавшись дать в нем некоторое представление о том, как соответствующие работы выглядят в более широком контексте искусства и науки программирования.

*Роберт Л. Эшснхёрст*  
*Редактор серии антологий*  
*ACM Press*

# Введение к части I

## ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Программирование является основой информатики. Программы описывают шаги вычислений, выполняемых вычислительными машинами, как физическими, так и абстрактными, и служат основой автоматического управления самыми разнообразными механизмами. На более высоком уровне программы используются для описания и сообщения алгоритмов всех видов с целью организовать сложные системы и управлять ими, а также избавить людей от множества утомительных и монотонных работ.

Языки программирования суть средства, с помощью которых выражаются почти все такие вычисления. Они играют двойную роль: это и системы обозначений, влияющие на наше мышление (см. лекцию Кеннета Айверсона 1979 г. в части II), и одновременно указания для абстрактной вычислительной машины, из которых инструкции для физической машины получаются в процессе автоматической трансляции. Поэтому неудивительно, что примерно половина Тьюринговских премий была присуждена за работы по языкам программирования, методологиям программирования и программным системам.

Десять лекций этой части книги были написаны за период 18 лет. Некоторые авторы подытожили в них свои технические достижения, идеи, которыми они руководствовались, и проблемные ситуации, в которых эти идеи возникли. Другие представили исследования, которыми они были заняты в то время, когда эти лекции читались. В совокупности все эти лекции составляют летопись многих важных достижений в информатике за период ее чрезвычайно интенсивного развития.

Читатель обнаружит в этих лекциях множество ссылок на работы других лауреатов Тьюринговской премии. Один из возможных способов организовать такую дискуссию — сгруппировать соответствующие материалы по тематическому признаку. Вместо этого я предпочла обсуждать эти работы в хронологическом порядке, чтобы дать некоторую историческую перспективу.

Алан Перлис прочел первую Тьюринговскую лекцию в 1966 г., когда большинство программ создавалось с помощью



кодовых бланков и перфокарт. Но А. Перлис смотрел в будущее. Признав ценность тьюринговской модели вычисления как оказавшей серьезное влияние на наше понимание сущности вычисления и важность влияния Алгола на наше мышление, Перлис обсуждает дальнейшие шаги, которые нужно сделать в области языков программирования и программных систем. Он объясняет, почему программист должен иметь возможность определять более богатый набор типов данных и структур данных вместе с соответствующими операциями над ними. Эту потребность впоследствии стремились удовлетворить исследования по абстрактным типам данных и продолжающиеся до сих пор исследования понятий о системах типов. Затем эти понятия стали важными составляющими систем Лисп и Смолток и играют важную роль в разработке программных сред. Хотя некоторые из поднятых Перлисом вопросов нашли успешное разрешение в последующих исследованиях, другие все еще не нашли удовлетворительного ответа.

Лауреатом Тьюринговской премии 1972 г. Эдсгер Дейкстра, который в то время был, пожалуй, наиболее известен своим письмом редактору журнала Communications of the ACM, в котором он подверг резкой критике оператор GOTO как «приглашение сделать из Вашей программы нечто совершенно невразумительное». Дейкстра делает ретроспективный анализ важнейших тем своей работы. Отметив значение систем EDSAC, языков Фортран, Алгол и Лисп, а также охарактеризовав полный язык PL/I как потенциально смертельное заблуждение, он выдвигает как главную проблему создание надежного программного обеспечения. Дейкстра развивает важную мысль о том, что ключом к созданию надежных программ является стремление избегать ошибок с самого начала, а не устранение их после того, как программа уже написана. Он доказывает, что безошибочное программирование и экономически важно, и технически осуществимо. Эта осуществимость может быть достигнута как результат ограничения интеллектуально постижимыми программами. Экономические доводы ныне хорошо известны.

Ко времени своего выступления с лекцией 1974 г. Дональд Кнут был уже широко известен своими достижениями в области программирования и программных систем, в том числе своей многотомной серией «Искусство программирования для ЭВМ». Кнут продолжал разрабатывать очень успешную систему высококачественного типографского набора TEX. Кнут сочетает любовь гуманитария к историческим корням современных идей с вниманием программиста к деталям. В своем обращении к программистской общественности он исследует историческую судьбу и происхождение понятий искусства и нау-

ки и доказывает, что программирование является одним из видов искусства. Хорошие программы могут и должны обладать изяществом и стилем. В духе истинного энтузиаста Кнут утверждает, что программирование должно доставлять наслаждение — вполне осмысленно писать программы ради одного лишь удовольствия от этого занятия. Наконец, он настаивает на создании «прекрасных» инструментов для программистов-художников и языков, поощряющих хороший стиль программирования.

Дана Скотт и Микаэль Рабин получили Тьюринговскую премию за их раннюю совместную работу по теории автоматов. Поскольку их области научных интересов впоследствии разошлись, они предпочли прочитать отдельные лекции о том, над чем каждый из них позже работал. Скотт впоследствии сделал фундаментальную работу в области математических оснований семантики языков программирования, развив теорию денотационной семантики. В своем выступлении он обратил внимание на основные идеи, лежащие в основе его работы. Он описывает хронологию своих личных профессиональных впечатлений, повлиявших на его подход к семантике, а затем переходит к самой работе, объяснив главную мысль и принципиальный результат, который он из этой мысли вывел. Работа Скотта остается важной и вызвала множество последующих исследований.

Хотя основной целью присуждения премии 1977 было признание работы Дж. Бэкуса по Фортрану и по описанию этого языка, Фортран был тогда известен программистам, и Бэкус уже давно обратил свое внимание на другие области исследования. Поэтому он воспользовался в своей лекции возможностью бросить вызов фундаментальному подходу фон-неймановского стиля программирования. Он доказывает, что традиционный стиль программирования приводит к затрате слишком больших усилий и времени на явное перемещение данных в оперативную память и из нее и на сложные правила присвоения имен и ограничения области действия операторов, которые поддерживают этот стиль. Вместо этого он предлагает стиль свободного от переменных функционального программирования, обладающего не только мощью и изяществом, но и математически обоснованной семантикой.

Статья Бэкуса обширнее его лекции и обращена не только к широкой программистской общественности, но и к специалистам. Неспециалистам может поначалу показаться пугающим объем новой для них нотации, но их усилия будут вознаграждены, даже если они лишь бегло просмотрят наиболее формальные разделы (но внимательно прочтут резюме в конце статьи!).

Тема, к которой обратился Роберт Флойд в лекции 1978 г., это парадигмы программирования. Флойд приводит много при-

меров, в том числе структурное программирование, рекурсивные сопрограммы, динамическое программирование, системы, основанные на правилах, и механизмы переходов между состояниями, а также показывает, как он использовал некоторые из этих парадигм в своих собственных исследованиях. Он доказывает, что важно сознательно относиться к применяемым парадигмам, учить им новичков и обеспечивать в языках программирования поддержку основных применяемых в программировании парадигм. Хотя за годы, прошедшие с тех пор, некоторые примеры могли измениться, но тема статьи Флойда остается столь же актуальной сегодня, как и в момент ее написания.

Я живо вспоминаю, как я слушала Тьюринговскую лекцию Чарлза Энтони Ричарда Хоара в октябре 1980 г., примерно год спустя после выпуска в свет языка Ада. Аудитория была словно заморожена, когда Хоар рассказывал об истории многих своих важных разработок в области языков и систем в контексте личных переживаний и поворотов своей профессиональной карьеры. Подытожив свой ранний опыт руководителя разработки программного обеспечения, он описывал свое участие в попытках создания языков, приведших к Алголу-68 и PL/I. Выстроив ряд примеров, иллюстрирующих преимущества простоты перед сложностью при разработке языков и систем, он связал этот свой опыт с проектом языка Ада, и высказал свое мнение о результате этого проекта. Читателю не следует упускать возможность познакомиться с историей, рассказанной Хоаром.

К тому времени, когда Кен Томпсон и Дэнис Ритчи были награждены Тьюринговской премией за 1983 г., система UNIX переместилась из исследовательской лаборатории на коммерческий рынок. В своей лекции Ритчи рассматривает факторы, способствовавшие успеху этой системы, включая долгий инкубационный период и создание системы в условиях, когда актуальность разработки важна, но коммерческое давление отсутствует. Ритчи приводит примеры других исследовательских разработок, в которых также была важна подобная питательная среда, и выражает озабоченность тем, что избыток актуальности может подавить и новаторство, и свободный обмен идеями.

Кен Томпсон прочел отдельную, но дополняющую выступление Ритчи лекцию. Как и ранее выступавшие Дейкстра, Хоар, Кнут и Флойд, Томпсон представил себя как программиста. Он разработал программу, которая начинается с примера, использованного Кнутом в его Тьюринговской лекции, и проиллюстрировал некую парадигму в смысле статьи Флойда, хотя это такая парадигма, которой Флойд может отказаться учить своих студентов! И Ритчи, и Томпсон дают остроумный социологиче-

ский комментарий той обстановке, в которой ныне разрабатываются программные системы.

Последняя из Тьюринговских лекций, помещенных в этой части книги, была прочитана Никлаусом Виртом в 1984 г. Вероятно, более всего известный как изобретатель Паскаля, Вирт имеет внушительный список признанных успешных разработок как создатель языков программирования. В рассказе о своей научной карьере он перечисляет важные разработки языков, в которых он принимал участие, и подчеркивает необходимость простоты. Поскольку его всегда интересовала эффективность реализации, он также обсуждает необходимость соответствия аппаратуры и программного обеспечения и излагает свой опыт разработки машины Лилит в качестве средства реализации языка Модула-2. Вирт указывает на преимущества непосредственного участия программиста в разработке и монтаже аппаратуры и на ценность правильного выбора инструментов.

Мне посчастливилось быть лично знакомой со всеми авторами, лекции которых представлены в этой части книги, а четверо из них были моими преподавателями в университете. Одно из удовольствий, испытанных мной при перечитывании этих лекций, — возможность увидеть отражения тех особых человеческих качеств, которые способствовали столь большому воздействию их работ. Нам действительно повезло, что у нас есть такие учителя, авторы, интеллектуальные лидеры и друзья.

*Сузан Л. Грэхем  
Беркли, Калифорния*

1966

# Синтез алгоритмических систем

*А. Дж. Перлис*

Институт технологии Карнеги  
Питсбург, Пенсильвания

## ВВЕДЕНИЕ

Знание и мудрость обогащают человека. Знание ведет к компьютерам, а мудрость — к китайским палочкам для еды. К сожалению, наша ассоциация слишком поглощена знанием. Мудрости придется подождать до более благоприятных дней.

На чем основывается и будет основываться слава Тьюринга? На том, что он доказал теорему о существовании для универсального вычислительного устройства — позднее названного машиной Тьюринга — функций, вычисление которых непосильно этому устройству? Я сомневаюсь в этом. Более вероятно, что она основывается на модели, которую он придумал и разрабатывал: на его формальном механизме.

Эта модель покорила воображение и завладела мыслями целого поколения ученых. Она обеспечила основы для споров, породивших теории. Его модель оказалась столь полезной, что пробудила активность не только математиков, но и представителей ряда инженерных дисциплин. Привлеченные ею доводы не всегда являются формальными, а последующие построения не всегда абстрактные. В сущности, самым плодотворным следствием появления машины Тьюринга стало построение, изучение и вычисление так называемых вычислимых функций, т. е. программирование для компьютеров. Это и неудивительно, потому что компьютеры способны вычислять намного больше того, что мы способны специфицировать.

Я уверен, что все согласится с тем, что эта модель имела исключительное значение. Историки простят меня за то, что я совсем не уделю в этой лекции внимания тому влиянию, которое Тьюринг оказал на разработку универсального цифрового компьютера, еще более ускорившего наше вовлечение в теорию и практику вычислений.

Хотя, разумеется, не только появление машины Тьюринга способствовало нашей вычислительной деятельности и поощряло ее. Я все же думаю, что столь же мощное воздействие оказал только один формальный механизм, называемый Алголом. Многие сразу возразят, указав, что среди нас слишком мало людей, которые понимали его или пользовались им. Хотя, к сожалению, они окажутся правы, но дело не в этом. Существенно

данный Алголом стимул к развитию исследований в информатике, а не число его последователей. К тому же Алгол мобилизовал наше мышление и обеспечил нас основой для дискуссий.

Я долго ломал себе голову над тем, почему Алгол оказался столь полезной моделью в нашей области деятельности. Возможно, к числу причин относятся следующие.

(а) Его международная поддержка.

(б) Ясность опубликованного описания его синтаксиса.

(в) Естественное сочетание в нем важных для программирования понятий сборки и подпрограммы.

(г) Тот факт, что язык можно естественным образом разбить на части, причем возможно предлагать и описывать довольно обширные модификации частей языка, не нарушая впечатляющую гармонию его структуры и нотации. Есть определенный смысл в термине «алголоподобный», который часто используется в рассуждениях о программировании, языках и вычислениях. Алгол, по-видимому, является жизнестойкой моделью и даже расцветает при хирургических вмешательствах — будь то обследования, пластические операции или ампутации.

(д) Тот факт, что он мучительно не приспособлен для многих задач, которые мы хотим программировать.

В одном я уверен: Алгол обязан своими магическими свойствами не процессу своего рождения, т. е. не тому, что это плод коллективного творчества. Итак, мы не должны разочаровываться, когда из таким же образом оплодотворенных яиц выходят более тупые модели. Эти последние, хотя и блещут впечатляющими улучшениями Алгола, вызывают у нас только скуку. Может быть, они и представляют собой улучшения Алгола, но не могут претендовать на роль преемников этой модели.

Конечно, мы должны и будем извлекать пользу из тех усовершенствований, которые позволяют исправлять недостатки Алгола. Нам следует подумать также и о том, почему они не могут стимулировать нашу творческую энергию. Нам следовало бы задаться вопросом, почему при их влиянии на научные исследования и даже на практику программирования они не обеспечивают качественного скачка вперед. Я не претендую на знание исчерпывающего ответа, но уверен, что значительная часть этой ограниченности обусловлена тем, что внимание сосредоточивается не на тех слабостях Алгола, на которые следует его обратить.

## СИНТЕЗ ЯЗЫКА И СТРУКТУР ДАННЫХ

Мы знаем, что проектируем язык, чтобы упростить выражение неограниченного количества алгоритмов, создаваемых для важного класса задач. Проектирование языка следует предпри-

нимать только тогда, когда алгоритмы для этого класса предполагают или, вероятно, обещают после некоторого развития значительную нагрузку на компьютеры, а программирование этих алгоритмов в случае использования существующих языков потребует значительных затрат времени. В такой ситуации новый язык должен сократить стоимость некоторого множества выполняемых работ, чтобы окупить затраты на свое проектирование, поддержание и усовершенствование.

Языки-преемники появляются по ряду причин.

(а) Исправление ошибки, недостаточности или избыточности в имеющемся языке *предполагает* естественное перепроектирование, результатом которого явится язык более высокого качества.

(б) Исправление ошибки, недостаточности или избыточности *требует* перепроектирования для получения полезного языка.

(в) На основе двух существующих языков обычно можно создать третий, который (1) содержит возможности обоих этих языков в интегрированной форме и (2) требует менее сложной грамматики и правил вывода, чем простое объединение грамматик и правил вывода, имеющихся в обоих языках.

Имея в виду сказанное выше, как можно было бы приступить к синтезу модели-преемника, которая не только улучшала бы коммерческие характеристики использования компьютеров, но и способствовала бы сосредоточению нашего внимания на важных проблемах внутри самого вычислительного процесса?

Я полагаю, что естественной отправной точкой должна быть организация и классификация данных. Было бы по меньшей мере затруднительно создать алгоритм, не зная природы обрабатываемых им данных. Когда мы пытаемся представить алгоритм на языке программирования, то должны узнать, как представляются на этом языке данные алгоритма, прежде чем сможем надеяться на полезные вычисления.

Поскольку наш преемник мыслится как универсальный язык программирования, он должен обладать общими структурами данных. В зависимости от точки зрения это может оказаться и не столь трудным, и не столь легким, как ожидается. Как следует организовать это представление структур данных? Посмотрим, что было сделано в уже имеющихся языках. Прежний подход состоял в следующем.

(а) В языке описывается несколько «примитивных» структур данных, например целые числа, вещественные числа, однородные по типу массивы, списки, строки и файлы.

(б) На этих структурах обеспечивается «достаточный» набор операций, например арифметические, логические, выделение, присваивание и комбинирование.

(в) Любые другие структуры данных считаются непримитивными и должны представляться через примитивные. Внутренняя организация в непримитивных структурах явно обеспечивается операциями над примитивными данными, например, взаимосвязь между вещественной и мнимой частями комплексного числа реализуется вещественной арифметикой.

(г) Достаточный набор операций для этих непримитивных данных организуется в виде процедур.

Этот процесс расширения нельзя обойти. Всякий язык программирования должен допускать легкое использование, поскольку в конечном счете это всегда требуется. Однако если злоупотреблять этим процессом расширения, то в алгоритмах часто исчезает та ясность структуры, которой они в действительности обладают. Что еще хуже, зачастую они неоправданно медленно выполняются. Первая слабость возникает потому, что для алгоритма язык был выбран неверно, а вторая — из-за того, что язык навязывает избыточную организацию данных и требует во время исполнения такого администрирования, которое могло бы быть выполнено однократно до исполнения алгоритма. В обоих случаях переменные связывались в неподходящее время правилами синтаксиса и вычисления.

Я думаю, все мы осознаем, что в наших языках недостаточно типов данных. Разумеется, в нашей модели преемника нам не следует пытаться возместить этот недостаток добавлением новых, например ограниченного количества новых типов и какой-то общей всеобъемлющей структуры.

Наш опыт описания функций должен подсказывать нам, что нужно делать: не сосредоточиваться на полном множестве описанных функций для общего пользования, а обеспечить в рамках языка такие структуры и управление, из которых следовало бы эффективное описание и использование функций в программе.

Поэтому нам нужно применительно к нашей модели преемника сконцентрировать внимание на обеспечении средств описания структур данных. Но этого недостаточно. В программе, для которой специфицируются структуры данных, должны быть еще представлены «достаточное» множество сопровождающих операций, контексты, в которых они встречаются, а также соответствующие правила вычисления.

Список некоторых возможностей, которые должны быть обеспечены для структур данных, мог бы включать:

- (а) описание структуры;
- (б) присваивание структуры идентификатору, т. е. придание идентификатору ячеек информации;
- (в) правила именования частей для заданной структуры;
- (г) присваивание значений для ячеек, отведенных для идентификатора;



(д) правила ссылок на ячейки, отведенные для идентификатора;

(е) правила комбинирования, копирования и стирания как структуры, так и содержимого ячеек.

Конечно, и теперь в большинстве языков эти возможности обеспечиваются в той или иной степени, но обычно слишком фиксированным способом, в виде правил синтаксиса и вычисления.

Мы знаем, что создатели языка не могут устанавливать, сколько информации должно находиться в структуре, а сколько — в данных, переносимых в этой структуре. Для каждой программы должен быть предоставлен естественный для нее выбор достижения баланса между временем и расходом памяти. Мы знаем, что не существует единого способа представления массивов или списковых структур, строк, файлов или их комбинаций. Выбор способа зависит от

(а) частоты доступа;

(б) частоты структурных изменений, в которых задействованы указанные данные, например присоединений к файлу новых структур записей или изменений границ массивов;

(в) затрат на излишнюю машинную память;

(г) затрат на излишнее время доступа к данным и

(д) важности алгоритмического представления, пригодного для упорядоченного наращивания, при котором всегда сохраняется ясность структуры.

Такой выбор, как всем известно, труден для программиста, а на уровне проектирования он совсем невозможен.

Структуры данных нельзя строить из воздуха. В сущности, наш обычный метод состоит в использовании базовой гипотетической машины с фиксированными примитивными структурами данных. Эти структуры — те, которые идентифицируются с реальными компьютерами, но базовая машина может быть более абстрактной применительно к описанию структур данных. После выбора базовой машины требуемые, согласно нашим описаниям, дополнительные структуры должны представляться как данные, т. е. как имя или указатель на структуру. Не все указатели ссылаются на однотипные структуры. Поскольку сегменты программы сами являются структурами, такие указатели, как «идентификатор процедуры с содержимым (x)», устанавливают класс переменных, значениями которых являются имена процедур.

## КОНСТАНТЫ И ПЕРЕМЕННЫЕ

На самом деле гибкость языка измеряется тем, что программистам разрешается изменять (как в композиции, так и

в исполнении). Систематическое развитие изменчивости в языке является центральной проблемой для программирования, а следовательно, и для проектирования нашего преемника. Всегда практика подсказывает нам особые случаи, на основании которых мы устанавливаем описания новых переменных. Каждый новый опыт сосредоточивает наше внимание на необходимости большей общности. Разделение времени является одним из новых явлений, которое, вероятно, станет повседневным. Разделение времени концентрирует наше внимание на управлении системами и на том, чтобы программисты управляли своими текстами до, во время и после их исполнения. Возрастает гибкость взаимодействия с программой, и наш преемник не должен затруднять этот процесс. Наше представление о диалоговом программировании означает много больше, чем быстрота и удобство отладки: наши наиболее интересные программы никогда не бывают совсем ошибочными, но и никогда не бывают окончательными версиями. Как программисты, мы должны выделить то новое, что вносит диалоговое программирование, прежде чем сможем надеяться обеспечить для него подходящую языковую модель. Я считаю, что новизна состоит в требовании изменяемости того, что прежде считалось фиксированным. Здесь я имею в виду не новые классы данных, а переменные, значениями которых являются программы или части программ, синтаксис или части синтаксиса и режимы управления.

Основное внимание теперь уделяется разработке систем управления файлами, которые улучшают управление всей системой, и относительно немного — улучшению управления вычислениями. Если первое может быть обеспечено помимо языков, на которых мы пишем свои программы, то для второго нам нужно упрочить свой контроль над изменчивостью в рамках языка программирования, используемого нами для решения своих проблем.

При обработке текста программы некий сегмент текста может встретиться один раз, но выполняться неоднократно. Поэтому возникает потребность идентифицировать и постоянство, и изменчивость. В общем случае мы берем то, что по форме является переменной, и делаем это постоянной в процессе инициализации; часто разрешается, чтобы сам этот процесс мог повторяться. Данный процесс инициализации является фундаментальным, и в нашем преемнике должен иметься систематический способ делать это.

Рассмотрим некоторые примеры инициализации и изменчивости в Алголе.

(а) *Вход в блок.* При входе в блок описания обеспечивают инициализацию, но только относительно некоторых свойств идентификаторов. Так, **integer** *x* инициализирует свойство при-

надлежности к типу целых чисел, но нет возможности инициализировать значения  $x$  как нечто, что не будет изменяться в области действия блока. Описание **procedure**  $P(\dots)$ ; ...; выражительно инициализирует идентификатор  $P$ , но нет возможности изменять его внутри блока. Описание **array**  $A[1:n, 1:m]$  присваивает начальную структуру. Нет возможности инициализировать значения ячеек этой структуры или изменять структуру, присвоенную идентификатору  $A$ .

(б) *Оператор цикла.* Не могут быть инициализированы выражения, которые я называю шаговым и конечным элементами.

(в) *Описание процедуры.* Инициализируется идентификатор процедуры. При вызове процедуры ее формальные параметры инициализируются, как и идентификаторы процедур, и они могут быть даже инициализированы по значению. Впрочем, разные обращения устанавливают различные инициализации идентификаторов формальных параметров, но не разные способы инициализации значений.

Предоставляемый в Алголе вариант связывания формы и содержания с идентификаторами считался подходящим. Однако если мы посмотрим на присваивание формы, вычисление формы и инициализацию как на важные функции, которые должны быть рационально специфицированы в языке, то можем счесть Алгол ограниченным и даже вычурным языком с точки зрения предоставляемого им выбора. Нужно надеяться на то, что язык-преемник окажется гораздо менее произвольным и ограниченным.

Я позволю себе привести тривиальный пример. В операторе цикла использование в качестве шагового элемента такой конструкции, как **value**  $E$ , где  $E$  — это выражение, должно было бы сигнализировать об инициализации выражения  $E$ . **value** — это разновидность оператора, управляющего связыванием значения с формой. Существует естественная область действия, соответствующая всякому применению этого оператора.

Я упоминал, что идентификаторы процедур инициализируются через описания. В таком случае связывание процедуры с идентификатором может быть изменено присваиванием. Я уже отмечал, как это делается с помощью указателей. Разумеется, существуют и другие способы. Простейший из них состоит в том, чтобы вовсе не заменять идентификатор, а пользоваться индексом выбора, который фиксирует одну процедуру из некоего множества. Теперь инициализация описывает массив форм, т. е. **procedure array**  $P[1:k](f_1, f_2, \dots, f_i); \dots \text{begin} \dots \dots \text{end}; \dots; \text{begin} \dots \text{end};$  Вызов  $P[j](a_1, a_2, \dots, a_j)$  выбрал бы для исполнения  $j$ -е тело процедуры. Или же можно описать **procedure switch**  $P: = A, B, C$  и процедурные указательные вы-

ражения, так что предыдущее обращение стало бы выбирать для исполнения  $j$ -е процедурное указательное выражение. Приведенные выше подходы оказываются слишком статичными для некоторых приложений, и им недостает важного свойства присваивания, а именно возможности определять, когда присвоенная форма не является более доступной, и следовательно, ее память можно использовать для иных целей. Возможными применениями таких динамически присваиваемых процедур являются генераторы. Предположим, у нас имеется процедура для вычисления (а)  $\sum_{k=0}^n C_k(N) X^k$  как приближенного значения некоторой функции (б)  $f(x) = \sum_{k=0}^{\infty} C_k X^k$  при специфицированной целой величине  $x$ . Теперь, найдя  $C_k(N)$ , мы заинтересованы только в вычислении (а) для различных значений  $x$ . Потом мы могли бы захотеть описать процедуру, которая подготавливает (а) на основании (б). Эта процедура при своем начальном выполнении присваивает либо себе самой, либо некоторому другому идентификатору процедуру, которая вычисляет (а). Последующие вызовы этого идентификатора только порождали бы созданное таким образом вычисление. Такое динамическое присваивание влекло бы за собой ряд привлекательных возможностей.

(а) В результате второго присваивания могла бы освободиться часть памяти программы.

(б) Память для данных может присваиваться как собственная (**own**) для идентификатора процедуры, описание которой создается.

(в) Начальный вызов может модифицировать результирующее описание; например, вызов по наименованию или вызов по значению формального параметра из исходного обращения может повлиять на вид полученного описания.

Легко видеть, что то, к чему мы пришли, — это необходимость единообразного подхода к инициализации и изменчивости формы и значения, относящихся к идентификаторам. В этом состоит требование вычислительного процесса. Сам по себе наш язык-преемник должен располагать общим способом управления действиями по инициализации и изменениям применительно к классам идентификаторов.

В частности, мы хотим выполнять в диалоговом программировании систематическое, или управляемое, изменение значений данных и текста в отличие от несистематических изменений, которые возникают в процессе отладки. Разумеется, выполнение таких действий означает, что определенные части текста понимаются как переменные. И снова мы достигаем этого с помощью определений, инициализации и присваивания. Поэтому мы можем писать в заголовке блока такие описания:

**real  $x, s$ ;**  
**arithmetic expression  $t, u$ ;**

В сопровождающем тексте появление оператора  $s := x + t$  вызывает сложение значения арифметического выражения, присвоенного переменной  $t$ , например, по вводу, со значением  $x$  и присваивание результата в качестве значения переменной  $s$ . Мы видим, что  $t$  может вводиться и храниться как некая форма. В таком случае операция  $+$  может выполняться только после применения подходящей функции преобразования. Нас не должно отпугивать то обстоятельство, что в классическое «время трансляции» может быть выполнена лишь частичная трансляция выражения. Настало время систематического подхода к проблеме частичной трансляции. Естественными изменяемыми частями текста являются те части, которые идентифицируются синтаксическими единицами языка.

Несколько более трудно принимать меры в связи с непредумышленным изменением программ. При этом основная проблема состоит в идентификации изменяемой части исходного текста и отыскании для нее соответствия в процессе трансляции с фактически вычисляемым текстом. Легко сказать: проинтерпретируй исходный текст. Но нужно найти удовлетворительный баланс затрат в рамках промежуточных решений между трансляцией и интерпретацией. Я надеюсь выразить в следующем разделе точку зрения, которая позволит пролить некоторый свет на достижение этого баланса в каждой программе, где это требуется.

## СТРУКТУРА ДАННЫХ И СИНТАКСИС

Даже если списковые структуры и рекурсивное управление не станут играть центральной роли в нашем языке-посреднике, он будет тесно связан с Лиспом. Этот язык вызывает забавные споры среди программистов, зачастую проклинающих и восхваляющих одни и те же особенности языка. Здесь я хотел бы только отметить, что описание Лиспа точно раскрывает соответствующие компоненты языковых средств с большей ясностью, чем в любом другом известном мне языке. Описание Лиспа включает не только его синтаксис, но и представление его синтаксиса и структуры данных среды в виде структуры данных языка. Фактически такое описание несколько ограничивает дальнейшее описание, но несущественно. На основании предшествующих описаний становится возможным описать процесс вычисления как программу на Лиспе с применением нескольких примитивных функций. Хотя такая завершенность

описания возможна и для других языков, в общем случае она не рассматривается как часть их определяющего описания.

Изучение Алгола показывает, что его структуры данных непригодны для представления текстов на Алголе, по крайней мере тем способом, который годится для описания вычислительной схемы языка. Аналогичное замечание относится и к его непригодности для описания внешних структур данных в программах на Алголе.

Я считаю критическим требование, чтобы наш язык-преемник обеспечил баланс структур данных, подходящих для представления синтаксиса и среды, так чтобы процесс вычисления можно было ясно сформулировать на этом языке.

Почему столь важно дать такое описание? Только ли для придания языку изящного свойства замкнутости, чтобы можно было организовать начальную загрузку? Вряд ли. Это является ключом к систематическому конструированию программных систем, пригодных для диалоговых вычислений.

Язык программирования характеризуется синтаксисом и набором правил вычислений. Последние связаны друг с другом посредством представления программ как данных, к которым применяют правила вычисления. Эта структура данных представляет собой внутренний или вычислительно ориентированный синтаксис языка. Мы составляем программы во внешнем синтаксисе, который фиксируется в целях общения между людьми. Внутренний синтаксис, как правило, считается столь связанным от машины и транслятора, что он почти никогда не описывается в литературе. Обычно существует процесс трансляции, который переводит текст из внешнего во внутреннее синтаксическое представление. Фактически изменение внутреннего описания более существенно связано с правилами вычисления, чем с машиной, на которой они должны выполняться. Выбор правил вычисления критически зависит от времени связывания переменных языка.

Тем самым указывается подход к организации вычислений, полезной в случае изменяемых текстов. Поскольку внутренняя структура данных отражает изменчивость обрабатываемого текста, пусть подходящее внутреннее представление синтаксиса выбирается в процессе трансляции и пусть общий вычислитель выбирает конкретные правила вычисления на основе предпочтенной синтаксической структуры. Итак, нам нужно задать во внешнем синтаксисе ключи, которые указывают переменные. Например, появление **arithmetic expression**  $t$ ; **real**  $u$ ,  $v$ ; и оператора  $u := v/3*t$ ; указывает возможность различного внутреннего синтаксиса для  $v/3$  и значения  $t$ . Следует отметить, что  $t$  по своему поведению весьма напоминает формальный параметр Алгола. Впрочем, здесь менее организован контроль над при-

сваиванием. Я думаю, что из этого следует только, что присваивания типа формальный параметр — фактический параметр не зависят от концепции замкнутой подпрограммы и что они объединены в конструкции процедуры как способ спецификации области действия инициализации.

В случае непредусмотренного заранее изменения программы знание внутренней синтаксической структуры позволяет свести к минимуму объем перетрансляции и изменения правил вычисления при изменении текста.

Поскольку нужно изучать и конструировать структуры данных и правила вычисления на некотором языке, представляется целесообразным, чтобы это был сам исходный язык. Можно определить в качестве цели трансляции внутренний синтаксис, в котором цепочки литер являются подмножествами текстов, допустимых в исходном языке. Если такой выбранный синтаксис близок к машинному коду, то его можно потом обрабатывать по правилам, весьма сходным с машинными.

Пока речь шла об изменяемости применительно к идентификаторам языка, и я ничего не сказал об изменяемости управления. В сущности, у нас нет способа описания управления, и поэтому мы не можем определять его организацию. Следует ожидать, что наш язык-преемник будет обладать некой формой управления наподобие принятой в Алголе — и не более того. Много исследований было посвящено такому виду управления, как параллельная работа. Кроме того, недавно в языках начало появляться распределенное управление, которое я буду называть мониторингом. Процесс *A* непрерывно осуществляет мониторинг процесса *B* таким образом, что, когда *B* достигает некоторого состояния, процесс *A* перехватывает управление дальнейшей деятельностью этого процесса. Управление в рамках процесса *A* может быть записано в виде **when *P* then *S***; *P* — это предикат, который проверяется всегда в пределах некоторого определенного контекста. Всякий раз, когда *P* есть истина, поднадзорное вычисление прерывается и выполняется *S*. Мы хотим механизировать эту конструкцию, проверяя *P* всякий раз, когда выполняется некое действие, которое, возможно, могло бы сделать *P* истиной, но ни в каких других случаях. В таком случае мы должны при описании языка, среды и правил вычислений включить состояния, которые могут подвергаться мониторингу во время исполнения. На основании этих примитивных состояний можно сконструировать посредством программирования и другие состояния. Зная эти примитивные состояния, можно предусматривать тестовые вставки в возможных точках еще до того, как конкретные предикаты определены в пределах программы. В таком случае мы можем диагностировать наши программы, не нарушая их целостности.

## ИЗМЕНЕНИЯ СИНТАКСИСА

В ограниченных рамках одного языка допустимы некоторые вариации. Однако весь наш опыт свидетельствует о том, что наши потребности в изменениях будут оказывать возрастающее давление на сам язык, стимулируя его изменения. Разработчики не в состоянии предугадать точные характеристики этих изменений, потому что те явятся следствиями еще ненаписанных программ для еще нерешенных задач. Увы, как раз наиболее полезные и успешные языки оказываются наиболее подверженными такому давлению. К счастью, довольно предсказуемы те изменения, которых нужно ожидать на ранних этапах. Например, в научных вычислениях представление чисел и арифметические действия над числами изменяются, но природа выражений не подвержена изменениям, кроме как через операнды и операции. Вызываемые этим модификации синтаксиса обозримы. В результате синтаксис и правила вычисления арифметических выражений остаются неопределенными в языке. Вместо этого в языке обеспечиваются правила синтаксиса и вычисления для программирования описаний арифметического выражения и для задания области действия таких описаний.

При этом единственная реальная трудность связана со спецификацией правил вычислений. При их задании следует соблюдать осторожность. Например, при введении таким образом арифметики матриц нужно вычислять матричные выражения с аккуратным использованием временной памяти, избегая необязательных итераций.

Естественный способ поупражняться в работе с описаниями состоит в том, чтобы начать с языка  $X$ , рассмотреть описания как расширение синтаксиса до синтаксиса языка  $X'$  и задать правила вычислений как процесс редукции, который преобразует любой текст на языке  $X'$  в эквивалентный текст на языке  $X$ .

Следует заметить, что изменение синтаксиса требует представления этого синтаксиса предпочтительно в виде структуры данных самого языка  $X$ .

## ЗАКЛЮЧЕНИЕ

Языки программирования строятся вокруг переменной — операций над ней, структур управления и данных. Поскольку эти понятия являются общими для всего программирования, общий язык должен сосредоточиваться на их последовательном развитии. Хотя мы в большом долгу перед Тьюрингом за предложенную им образцовую модель, нас не смущает работа с машинами и данными, превосходящими по своей сложности тот



уровень, который Тьюринг считал необходимым. Программисты никогда не удовлетворятся языками, позволяющими им программировать что угодно, но не позволяющими удобно и легко делать то, что их действительно интересует. Поэтому наш прогресс оценивается достигаемым нами балансом между эффективностью и общностью. С изменением характера использования вычислений (а это происходит) изменяется пригодное для наших целей определение языка и наиболее актуальными становятся иные проблемы. Я предчувствую, что наша модель преемника будет отражать такое изменение. Информатика — непоседливое дитя; ее прогресс столь же зависит от изменений в подходе, как и от последовательного развития наших современных концепций.

Ни одна из представленных здесь идей не нова; просто время от времени о них забывали.

Хочу поблагодарить Ассоциацию за предоставленную мне честь прочитать эту первую Тьюринговскую лекцию. И лучше всего ее закончить словами, что, если бы Тьюринг был сегодня с нами, он сказал бы о другом в лекции с иным названием.

## ПОСТСКРИПТУМ

А. Дж. Перлис  
Факультет информатики  
Йельский университет

В столь динамично развивающейся проблематике, как программирование, не следует ожидать, что статья двадцатилетней давности останется последним словом науки. Поэтому я был приятно удивлен, обнаружив очевидные интерпретации ее содержания, согласующиеся с тем, что происходило и все еще происходит с языками программирования.

Мы по-прежнему придаем огромное значение модели вычислений, которая предстает перед нами в облике языка программирования. Большинство новых языков, овладевших нашим воображением, придают этим моделям сладость синтаксиса и обогащают их семантику, так что теперь нас соблазняют такие честолюбивые эксперименты, которые мы зарекались проводить прежде. Рассмотрим четыре такие модели: конвейеры (APL), распределенное программирование (более известное под названием объектно-ориентированное программирование, примером может служить язык Смолток), редукционное программирование (функциональное программирование, примеры — языки Лисп, ЕР и ML) и непроецедурное программирование (например, логическое программирование на Прологе). Эти модели завладели нашими умами примерно так же, как Алгол 25 лет назад. У нас нет оснований предполагать, что это последние модели, способные побудить нас попытаться достичь еще более грандиозных обобщений.

Моя лекция была сосредоточена на важности структур данных в программировании, а следовательно, в языке программирования. Одно из направлений развития языков состоит в возрастающем усложнении описаний и управления структурами данных. Причем Алгол и его прямые «отпрыски» ориентировались на четко определенную изменяемость структур данных, ведущую к записям и к теориям типов, а упомянутые нами выше модели ориентировались на свою зависимость от одной составной структуры данных, напри-

мер списка или массива. Разумеется, по мере расширения их использования забота об изменчивости структур данных этих моделей должна **возрасти**.

Рабочая станция, персональный компьютер и сеть тогда еще не стали общепринятым инструментарием, в который надлежало интегрировать системы языков программирования. Редакторы были примитивными и **никоим** образом не выглядели как волшебная дверь, через которую мы попадаем в страну вычислений. Тем не менее в лекции нашла отражение важность диалоговых вычислений и языков для поддержки таких вычислений. Было отмечено, что представление программ как данных являлось критическим для подобного программирования. Оказывается, что для такого **представления** списки подходят лучше, чем массивы, поскольку синтаксис представляет собой набор вложенных и подставляемых ограничений, что совпадает с правилами модификации списковых структур. Язык APL и конвейерная организация пострадали из-за этого неравноправия между массивами и списками. В новых версиях APL предпринимаются попытки преодолеть это **неравноправие**.

Программирование становится повсеместной деятельностью, и предпринимаются энергичные усилия, чтобы стандартизировать небольшое количество языков (моделей и средств). До сих пор мы сопротивлялись такому ограничению, и это было мудро с нашей стороны. Новые архитектуры и проблемные области, безусловно, подскажут новые вычислительные модели, которые потрясут наше воображение. От них и от нынешнего состояния **про**граммирования произойдет новое видение языка, играющего роль проекта Ада. Как всегда, мы по-прежнему будем избегать ловушки Тьюринга: **необ**ходимости пользоваться языками, которые позволяют делать все, но не делают простым и удобным ничего из того, в чем мы действительно **заинтересованы**.

1972

## Смиренный программист

*Эдсгер В. Дейкстра*

Отрывок из объявления о награждении Тьюринговской премией, зачитанного М. Д. Макилроем, председателем Комитета по Тьюринговским премиям во время представления настоящей лекции 14 августа 1972 г. на ежегодной конференции АСМ в Бостоне:

«Повсюду профессиональный словарь программиста полон слов, введенных или предложенных Э. В. Дейкстрой, — дисплей, мертвая хватка, семафор, программирование с минимумом операторов «go to», структурное программирование. Однако его влияние на программирование является более всепроникающим, чем это мог бы подсказать любой словарь. Ценнейший вклад Дейкстры, признанием которого служит эта Тьюринговская премия, — его стиль: подход к программированию как к высокому искусству и интеллектуальному творчеству, настойчивые требования и практическая демонстрация того, что программы должны быть с самого начала правильно составлены, а не просто отлаживаться до тех пор, пока не станут правильными; ясное понимание того, какие проблемы лежат в основе программирования. Он опубликовал около дюжины статей как технического, так и концептуального характера, среди которых следует особенно отметить его философские обращения к IFIP [1], а также ставшие классическими статьи о взаимодействующих последовательных процессах [2], и его знаменитый обвинительный акт против оператора go to [3]. Недавно появились в виде элегантной монографии оказавшие большое влияние эссе Дейкстры об искусстве составления программ [4]».

Мы научились оценивать хорошие программы так же, как мы оцениваем хорошую литературу. И в центре этого движения находится Э. В. Дейкстра, который создает образцы, столь же прекрасные, как и полезные, и размышляет над ними

Вследствие длинной череды совпадений я официально стал программистом первым весенним утром 1952 г. и, насколько мне удалось выяснить, был первым голландцем, начавшим заниматься этим в моей стране. По прошествии времени я нахожу, что самая поразительная вещь во всем этом, по крайней мере в моей части света, — это та замедленность, с которой появлялась профессия программиста, неторопливость, в которую сейчас трудно поверить. Но, к счастью, в моей памяти сохранилось два ярких воспоминания о том времени, которые не оставляют никаких сомнений в этой неторопливости.

После того как я прозанимался программированием где-то около трех лет, у меня состоялась важная беседа с ван Вейнгаарденом, который был тогда моим руководителем в Амстердамском математическом центре, — беседа, за которую я буду благодарен ему, пока жив. Дело было в том, что, как предпо-

лагалось, я должен был параллельно изучать теоретическую физику в Лейденском университете, но совмещать эти два занятия становилось все труднее, и мне надо было сделать выбор — либо прекратить программировать и стать настоящим уважаемым теоретическим физиком, либо как-то формально завершить мое обучение теоретической физике с минимальными усилиями и стать... кем же? Программистом? Но разве это уважаемая профессия? В конце концов, что такое программирование? В чем должен был состоять тот солидный объем знаний, который позволил бы считать программирование достойной уважения интеллектуальной научной дисциплиной? Я живо вспоминаю, как я завидовал своим коллегам-электронщикам, которые в ответ на вопрос об их профессиональной компетенции могли по меньшей мере указать, что они знают все об электровакуумных приборах, усилителях и прочем, в то время как я чувствовал, что мне нечего ответить на подобный вопрос. Полный мрачных предчувствий, я постучал в дверь кабинета ван Вейнгаардена и спросил, может ли он уделить мне несколько минут для разговора. Когда я покидал его кабинет несколько часов спустя, я был другим человеком. Терпеливо выслушав, что меня заботит, он согласился, что в настоящее время есть не так уже много вещей, которые можно было бы отнести к дисциплине программирования, но затем он спокойно продолжал объяснять, что автоматические вычислительные машины — не кратковременная мода, за ними будущее, что мы находимся у самых истоков и — как знать? — может быть, именно я призван в будущем превратить программирование в почтенную научную дисциплину. Это был поворотный момент всей моей жизни, и я завершил свои занятия теоретической физикой настолько быстро, насколько это было возможно. Из этой истории следует по крайней мере один вывод: давая советы молодежи, нужно быть очень осмотрительным: иногда молодые следуют этим советам!

Два года спустя, в 1957 г., я женился, а в Голландии при регистрации брака необходимо отвечать на вопрос о вашей профессии, и я заявил, что я программист. Но муниципальные власти города Амстердама это не устроило — они считали, что такой профессии не существует. И хотите верьте, хотите нет — в графе «профессия» в моем брачном свидетельстве стоит смехотворная запись: «физик-теоретик»!

Этого достаточно, чтобы показать, насколько медленно профессия программиста завоевывала признание в моей стране. С тех пор я повидал немало других стран, и мое общее впечатление состоит в том, что и там, кроме, возможно, некоторого сдвига по времени, процесс становления этой профессии был очень похожим.

Позвольте мне описать положение в те давние времена несколько подробнее; надеюсь, это поможет лучше понять современную ситуацию. Попутно мы увидим, как много недоразумений относительно истинной природы программирования уходят своими корнями в то теперь уже далекое прошлое.

Все первые автоматические электронные компьютеры были уникальными, построенными в единственном экземпляре машинами, и окружение, в котором все они были сооружены, отличала волнующая атмосфера экспериментальной лаборатории. Как только возникала идея построить автоматический компьютер, ее реализация становилась дерзким вызовом доступной в те времена электронной технологии, и одно можно сказать наверняка: мы не можем отказывать в смелости тем людям, которые решались попробовать создать такую фантастическую технику. А она действительно была фантастической: теперь можно только удивляться, что те первые машины вообще работали, хотя бы изредка. Надо всем царила одна цель — привести компьютер в рабочее состояние и поддерживать его в этом состоянии. Озабоченность физическими аспектами автоматического вычисления до сих пор отражена в названиях самых старых научных обществ в этой области, таких, как Ассоциация вычислительных машин (АСМ) или Британское вычислительное общество; в этих названиях прямо содержится упоминание об аппаратной части.

А как же несчастный программист? Честно говоря, его едва замечали. И все потому, что первые машины были настолько громоздки, что их даже невозможно было сдвинуть с места, а кроме того, они требовали такого трудоемкого обслуживания, что было вполне естественно пытаться использовать машину в той же лаборатории, в которой она была разработана. Во-вторых, невидимая в каком-то смысле работа программиста была лишена всякого внешнего блеска: посетителям можно было демонстрировать машину, а это на много порядков превышает по зрелищности какой-то клочок бумаги с программой. Но, что важнее всего, сам программист относился к своей работе как к весьма скромному делу: вся его значительность была связана с существованием этой замечательной машины. Поскольку это была единственная машина, он слишком хорошо знал, что его программа годится лишь для нее одной, и так как было очевидно, что срок существования этой машины ограничен, он знал, что очень немногое в его работе имеет непреходящее значение. И наконец, было еще одно обстоятельство, оказавшее глубокое влияние на отношение программиста к своей работе: с одной стороны, кроме того, что его машина ненадежна, она еще и слишком медленна, а ее память слишком мала, т. е. ресурсов всегда недостаточно, а с другой стороны, ее обычно не-

сколько странный код команд допускал самые неожиданные конструкции. И в те дни многие умные программисты получали огромное интеллектуальное удовольствие от хитрых приемов, посредством которых им удавалось добиваться невозможного, разрешая довольно сложные задачи при всех тех ограничениях, которые налагало оборудование.

С тех времен берут начало два распространенных мнения о природе программирования. Я сейчас их упомяну и вернусь к ним позже. Первое мнение состояло в том, что по-настоящему компетентный программист должен обладать склонностью к разгадыванию головоломок и любить хитроумные уловки; другое в том, что программирование есть не более чем та или иная оптимизация эффективности вычислительного процесса.

Последнее мнение явилось результатом часто возникавшей ситуации: действительно, доступное оборудование нередко сурово ограничивало возможности вычислений, и в те времена часто приходилось сталкиваться с наивной надеждой, что, как только появятся более мощные машины, программирование перестанет быть проблемой, потому что тогда не будет необходимости выжимать из машины все, что она может. А ведь программирование как раз в этом и состоит, не правда ли? Но в следующие десятилетия произошло нечто совсем другое: стали доступными более мощные машины, более мощные даже не на один порядок, а на несколько. Но вместо того, чтобы найти все проблемы программирования решенными раз и навсегда, мы поняли, что наступил настоящий кризис программного обеспечения. Почему же так получилось?

Есть одна несущественная причина: в одном или двух отношениях современная аппаратура принципиально более сложна в обращении, чем старая. Во-первых, появились прерывания ввода-вывода, они происходят непредсказуемо, и моменты их появления невоспроизводимы. По сравнению со старой последовательной машиной, которая по замыслу являлась полностью детерминированным автоматом, такая разница драматична, и седые волосы многих системных программистов свидетельствуют о том, что мы не должны легкомысленно относиться к возникающим при этом логическим проблемам. Во-вторых, современные машины оборудованы многоуровневыми запоминающими устройствами, что заставляет нас думать о стратегии управления, которая, несмотря на обширную литературу на эту тему, остается весьма неясной. Вот, пожалуй, и все, что стоит сказать о структурных новшествах современных машин.

Я назвал все это несущественной причиной, главная же причина в том, что ... машины стали на много порядков более мощными! Подойдем к этому весьма прямолинейно: когда машин не было вовсе, программирование не составляло никакой проб-

лемы; когда у нас было несколько маломощных компьютеров, программирование стало проблемой средней сложности, а теперь, когда мы располагаем гигантскими компьютерами, программирование в свою очередь превращается в гигантскую проблему. В этом смысле электронная промышленность не разрешила ни одной проблемы, она их только создала, а именно проблему использования своей продукции. Другими словами, по мере того как мощность машин возрастает более чем в тысячу раз, честолюбивые замыслы общества, связанные с применением этих машин, растут в той же пропорции, и именно несчастный программист обнаруживает, что его работа оказалась в поле напряжения между целями и средствами. Возросшая мощность аппаратного обеспечения вместе с, возможно, еще более драматичным ростом надежности оборудования, сделали приемлемыми решения, о которых программист раньше не осмеливался и мечтать. А теперь, несколько лет спустя, он вынужден мечтать о них, и, что еще хуже, он должен сделать подобные мечты реальностью! Удивительно ли, что мы пришли к кризису программного обеспечения? Разумеется, нет, и, как легко догадаться, это предсказывалось заранее. Однако с предсказаниями не очень крупных пророков всегда бывает так: лишь через лет пять всем становится ясно, что они были верными.

Тогда, в середине шестидесятых, случилось нечто ужасное: появились компьютеры так называемого третьего поколения. Из официальных документов известно, что отношение цена/качество было одной из главных целей их разработки. Но если считать «качеством» коэффициент использования различных компонентов машины, вы едва ли сумеете избежать такого проекта, в котором «качество» достигается в основном за счет внутренних обменов информацией между компонентами машины, полезность которых сомнительна. А если ваше определение цены означает цену, которую необходимо заплатить за аппаратуру, то у вас, скорее всего, получится такой компьютер, для которого будет очень трудно программировать. Например, набор команд мог бы быть таким, что уже на ранних этапах на программиста или систему будут наложены ограничения, приводящие к неразрешимым на самом деле конфликтам. И похоже, в значительной степени подобные неприятные возможности становятся реальностью.

Когда о таких ЭВМ было объявлено и стали известны их функциональные спецификации, многим из нас, должно быть, стало не по себе; по крайней мере такое чувство возникло у меня. Было естественно ожидать, что такие вычислительные машины хлынут потоком на компьютерный рынок, и, следовательно, тем более важно, чтобы их конструкция была как можно более разумной. Но проект содержал такие серьезные ошиб-

ки, что я почувствовал, что одним ударом прогресс в информатике был заторможен по меньшей мере на десять лет. Это была самая черная неделя во всей моей профессиональной карьере. Быть может, печальнее всего то, что после всех этих лет разочаровывающего опыта многие все еще честно верят в то, что в силу некоторого закона природы машины должны быть именно такими. Они заглушают свои сомнения, видя, как много таких машин было продано, и выводят отсюда, что в конце концов их конструкция не должна быть уж очень скверной. Но при более пристальном рассмотрении подобная линия защиты имеет такую же убедительность, как вывод о том, что курение — здоровое занятие, ведь так много людей курят.

Именно в этой связи я сожалею, что научные журналы в области информатики не имеют обыкновения публиковать обзоры новых проектов компьютеров, вроде того как они печатают обзоры научных публикаций. Обзор вычислительных машин был бы не менее важен. И вот я хотел бы здесь покаяться: в самом начале шестидесятых годов я написал такой обзор и намеревался его отправить в журнал «Communications of ACM», но несмотря на то, что несколько коллег, которым я разослал текст обзора для отзыва, советовали мне это сделать, я не осмелился, боясь, что трудности как для меня, так и для редколлегии журнала могут оказаться слишком большими. Это было с моей стороны актом трусости, и за это я себя ругаю все больше и больше. Те трудности, которые я предвидел, были следствием отсутствия общепризнанных критериев, и хотя я был убежден в справедливости предпочтенных мною критериев, я опасался, что мой обзор не будет принят или будет забракован как «отражающий» лишь личное мнение. Я до сих пор полагаю, что подобные обзоры были бы крайне полезны, и с нетерпением ожидаю, когда они появятся, поскольку их признание было бы верным знаком зрелости программистской общественности.

Причиной, по которой я уделил столько внимания аппаратной части, было ощущение, что одним из наиболее важных аспектов любого вычислительного устройства является его влияние на способ мышления тех, кто его использует. У меня есть причины полагать, что это влияние во много раз сильнее, чем обычно думают. Переключим теперь наше внимание на программное обеспечение.

Вначале была машина EDSAC в Кембридже, Англия, и я считаю очень примечательным, что с самого начала понятие библиотеки подпрограмм играло центральную роль в конструкции этой машины и способе, которым она должна была использоваться. Теперь прошло 25 лет, и все, что связано с компьютерами, изменилось самым драматичным образом, однако понятие базового программного обеспечения до сих пор принас,



а понятие замкнутой подпрограммы до сих пор одно из ключевых в программировании. Мы должны признать, что замкнутая подпрограмма является одним из величайших изобретений программного обеспечения. Оно пережило три поколения компьютеров и переживет еще больше, потому что воплощает одну из наших фундаментальных абстракций. К сожалению, важность подпрограммы недооценивалась при создании третьего поколения компьютеров, в котором большое число явно присвоенных имен регистров арифметического устройства приводит к большим непроизводительным затратам при реализации подпрограмм. Но даже это не погубило понятие подпрограммы, и нам только остается молиться о том, чтобы эта мутация не передавалась по наследству.

Второе крупное преобразование в области программного обеспечения, о котором мне хотелось бы напомнить, — это рождение Фортрана. В те времена это был чрезвычайно смелый проект, и разработавшие его люди заслуживают нашего восхищения. Было бы абсолютно несправедливо винить их в недостатках, которые выявились только примерно после десяти лет его широкого использования: ведь коллективы разработчиков, которым удастся предвидеть на десять лет вперед, чрезвычайно редко встречаются! В ретроспективе мы должны оценивать Фортран как успешную методику программирования, но с очень ограниченным числом средств поддержки основной концепции, — средств, которые ныне столь остро необходимы, что пришла пора считать этот язык устаревшим. Чем раньше мы забудем, что Фортран когда-либо существовал, тем лучше, потому что в качестве способа мышления он уже перестал быть адекватным: он заставляет нас напрасно тратить наши мыслительные способности; кроме того, он слишком рискован, и, следовательно, им слишком дорого пользоваться. Трагическая судьба Фортрана — следствие его широкого признания, приковавшего мышление тысяч и тысяч программистов к нашим прошлым ошибкам. Я денно и нощно молю, чтобы как можно больше моих собратьев-программистов нашли способ освободиться от проклятия совместимости.

Третий проект, о котором мне не хотелось бы умолчать, это Лисп, вдохновляющее предприятие совсем другого рода. При небольшом числе весьма фундаментальных принципов он проявил замечательную устойчивость. Кроме того, на использовании Лиспа основаны многие в некотором смысле наиболее изощренные программные продукты. В шутку Лисп описывался как «наиболее интеллигентный способ злоупотребления компьютером». Я думаю, что подобная характеристика является большим комплиментом, поскольку она передает всю полноту освобождения: Лисп помогает многим из наших наиболее уда-

ренных коллег мыслить о вещах, ранее считавшихся немислимыми.

Четвертый проект, который хотелось бы упомянуть, — это Алгол-60. В то время как вплоть до нынешних дней программисты на Фортране склонны понимать свой язык программирования в терминах тех специфических реализаций, над которыми они работают (отсюда и преобладание восьмеричных или шестнадцатеричных распечаток), а определение Лиспа до сих пор остается причудливой мешаниной из того, что язык означает, и того, как он работает, знаменитое Сообщение об алгоритмическом языке Алгол-60 является плодом подлинных усилий перейти на следующий уровень абстрактности и определить язык программирования способом, не зависящим от его реализации. Критически настроенные могли бы сказать, что авторы Сообщения настолько в этом преуспели, что посеяли серьезные сомнения в самой возможности его реализации! Сообщение великолепно продемонстрировало могущество формального метода БНФ, хорошо известной ныне формы Бэкуса — Наура, а также могущество тщательно сформулированной английской прозы, по крайней мере когда ею пользуется кто-то столь же талантливый, как Питер Наур. Я полагал, что справедливости ради следует сказать: только очень небольшое число столь кратких, как этот, документов имели столь глубокое влияние на специалистов по информатике. Пожалуй, сомнительным комплиментом репутации Алгола послужила та легкость, с которой в более поздние времена использовались слова «Алгол» и «алголоподобный»; подобно незащищенному товарному знаку, они одаживали свою славу порой совсем не имеющим к ним отношения незрелым проектам. Сила метода БНФ как инструмента для определения ответственна за то, что я рассматриваю как одну из слабых сторон этого языка: излишне сложный и не слишком систематический синтаксис стало возможным втиснуть в несколько страниц текста определений. Такой мощный инструмент, как БНФ, мог бы сделать сообщение об алгоритмическом языке Алгол-60 намного короче. Кроме того, у меня начали возникать сомнения по поводу механизма параметров в Алголе-60; он дает программисту так много комбинационной свободы, что для того, чтобы программист пользовался им с уверенностью, он должен соблюдать жесткую дисциплину. Кроме того, что его реализация обходится слишком дорого, им, по-видимому, опасно пользоваться.

Наконец, хотя следующий сюжет и не слишком приятен, я должен упомянуть PL/I, язык программирования, для которого определяющая его документация имеет устрашающий объем и сложность. Пользоваться языком PL/I, должно быть, так же сложно, как управлять самолетом с 7000 кнопок, переключате-

лей и рукояток в кабине пилота. Я совершенно неспособен понять, как мы можем надеяться сохранять ориентацию в наших постоянно растущих программах, если из-за своей явной вычурности язык программирования — напомним, что это наш основной инструмент! — уже ускользнул из-под нашего контроля. И если бы мне надо было описать то влияние, которое имеет язык PL/I на своих пользователей, наиболее подходящая метафора, которая приходит мне на ум, — это наркотик. Я вспоминаю одну лекцию на симпозиуме по языкам программирования высокого уровня, прочитанную в защиту языка PL/I человеком, называвшим себя его преданным пользователем. Но в конце этой часовой лекции, восхваляющей PL/I, он ухитрился попросить добавления около 50 новых «свойств», ничуть не догадываясь, что основным источником его трудностей является именно то, что язык уже имеет слишком много «свойств». Лектор проявил все удручающие симптомы наркомании: дошедший до отупления, он мог лишь просить все больше, больше, больше... В то время как Фортран был назван детской болезнью, весь язык PL/I, растущий с неудержимостью опасной опухоли, может оказаться неизлечимой болезнью.

Вот что было в прошлом. Однако нет никакого смысла делать ошибки, если потом не учиться на них. В действительности же я считаю, что мы столь многому уже научились, что через несколько лет программирование будет настолько существенно отличаться от того, что оно представляло собой до сих пор, что лучше бы было заранее подготовиться к этому удару. Я попробую набросать здесь один из вариантов будущего. При первом взгляде на нее эта картина программирования в недалеком теперь уже будущем может поразить вас как крайне фантастическая. Я хочу добавить несколько соображений, которые помогут поверить в то, что она может стать реальностью.

А картина такова, что еще до завершения семидесятых мы сможем изобрести и реализовать системы такого рода, которые сейчас находятся на пределе наших возможностей программировать, и затратить на них лишь несколько процентов тех усилий, которых они требуют ныне, и, кроме того, эти системы будут практически свободными от ошибок. Эти два усовершенствования будут идти рука об руку. В последнем отношении программное обеспечение, по-видимому, отличается от многих других продуктов, когда более высокое качество, как правило, связано с более высокой ценой. Те, кто желает иметь действительно надежное программное обеспечение, обнаружат, что они для начала должны найти способ избежать большинства ошибок, и как результат, процесс программирования станет дешевле. Если вы желаете иметь более эффективных программистов, вы обнаружите, что они не должны терять время на отладку

и устранение неполадок, они прежде всего не должны делать ошибок. Другими словами, достижение обеих этих целей требует одного и того же изменения подхода.

Такая решительная перемена за такой короткий период времени была бы революцией, и всем, кто основывает свои представления о будущем на гладких экстраполяциях недавнего прошлого, ссылаясь на некие неписанные законы социальной и культурной инерции, вероятность такой решительной перемены может показаться пренебрежимо малой. Но мы все знаем, что революции иногда происходят. Но насколько вероятно, что эта революция произойдет?

По-видимому, надо, чтобы выполнились три основных условия. Весь мир должен признать необходимость перемены; во-вторых, экономическая необходимость этой перемены должна быть достаточно сильной; и, в-третьих, эта перемена должна быть технически выполнимой. Обсудим эти три условия в перечисленном порядке.

Что касается признания необходимости большей надежности программного обеспечения, то я не ожидаю никаких возражений. Но не столь давно дело обстояло иначе: разговор о кризисе программного обеспечения считался кощунством. Поворотной точкой была Конференция по технике программного обеспечения в Гармише в октябре 1968 г. Эта конференция стала сенсационной, когда на ней впервые было открыто признан кризис программного обеспечения. А теперь повсеместно признано, что разработка какой-либо крупной сложной системы является трудным делом и люди, ответственные за такие предприятия, обычно очень озабочены именно вопросами надежности, и это вполне справедливо. Короче говоря, наше первое условие, по-видимому, выполнено.

Что касается экономической необходимости, то прежде можно было зачастую встретить с мнением, что в шестидесятые годы программистам платили слишком много и в будущем их заработки должны упасть. Обычно это мнение высказывается в связи с экономическим спадом, но оно может быть и симптомом какого-нибудь другого и вполне здорового явления, а именно: программисты прошлого десятилетия, похоже, не столь хорошо выполняли свою работу, как были должны. В обществе нарастает неудовлетворенность качеством работы программистов и их продуктов. Но имеется еще один фактор, существенно более важный. При теперешнем положении дел вполне обычно, что для конкретной системы цена, назначаемая за разработку программного обеспечения, имеет такой же порядок величины, как цена аппаратуры, и общество более или менее спокойно принимает это. Но изготовители аппаратуры сообщают нам, что, как ожидается, в следующем десятилетии цены на

аппаратуру упадут в десять раз. Если разработка программного обеспечения будет оставаться таким же громоздким и дорогостоящим процессом, как ныне, это равновесие цен резко нарушится. Нельзя ожидать, что общество с этим согласится, и, следовательно, нам надо научиться программировать на порядок эффективнее. Иначе говоря, пока вычислительные машины были самой крупной статьей расходов, программистам удавалось получать финансирование, несмотря на их неуклюжую и расточительную методологию, но это прикрытие исчезнет очень быстро. Короче говоря, второе наше условие тоже выполнено.

Теперь рассмотрим третье условие. Является ли такая перемена технически выполнимой? Я думаю, что это возможно, и приведу вам шесть доводов в пользу этого моего мнения.

Изучение структуры программ показало, что программы — даже альтернативные программы для одной и той же задачи и с тем же математическим содержанием — могут очень сильно различаться по своей удобочитаемости и предсказуемости. Было открыто несколько правил, нарушение которых либо серьезно нарушает, либо полностью разрушает удобочитаемость и предсказуемость поведения программы. Имеется два рода правил. Выполнение правил первого рода нетрудно обеспечить механически, т. е. подходящим выбором языка программирования. Примером служит исключение операторов `go to` и процедур с более чем одним внешним параметром. Что касается правил второго рода, то я по крайней мере — возможно, из-за отсутствия компетентности — не вижу способов их механического установления, поскольку, по-видимому, требуется некое устройство для доказательства теорем; неизвестно, можно ли обеспечить такой механизм. Таким образом, в настоящее время, а может быть, и навсегда, правила второго рода представляют собой элементы той дисциплины, которая требуется от программиста. Некоторые из правил, которые я имею в виду, настолько очевидны, что им легко обучить, и не может возникнуть споров по поводу того, удовлетворяет ли им данная программа или нет. Примерами служат требования, что нельзя написать цикла, не обеспечив доказательства правила остановки или без формулирования соотношения, которое остается неизменным, сколько бы раз не выполнялись повторяемые операторы.

Я предлагаю ограничиться пока разработкой и реализацией удобочитаемых программ с предсказуемым поведением. Если кто-то опасается, что это ограничение слишком сурово и мы не можем с ним мириться, я могу его уверить: класс удобочитаемых и предсказуемых программ все еще достаточно широк и содержит много очень реалистичных программ для любой алгоритмически разрешимой задачи. Мы не должны забывать, что составлять программы — не наше дело: наше дело разра-

батывать классы вычислений, ведущих себя предсказуемым образом. Мое предложение ограничиться удобочитаемыми и предсказуемыми программами — это основа для первых двух из шести объявленных мной доводов.

Первый довод состоит в том, что если программисту приходится рассматривать только программы с предсказуемым поведением, то альтернативы, из которых он выбирает, намного легче оценивать и сравнивать.

Второй довод заключается в том, что, поскольку мы решили ограничиться предсказуемыми программами, мы раз и навсегда добились резкого сокращения рассматриваемого пространства решений. И этот довод отличен от предыдущего.

Третий довод основан на конструктивном подходе к проблеме правильности программы. В настоящее время общепринятой техникой является составление программы, а затем ее тестирование. Однако тестирование программы может быть очень эффективным способом демонстрации наличия ошибок, но оно безнадежно неадекватно для доказательства их отсутствия. Единственный эффективный способ значительно повысить доверие к программе — это дать убедительное доказательство ее правильности. Но не следует сначала писать программу, а потом доказывать ее правильность, поскольку в этом случае требование найти доказательство только увеличит тяготы бедного программиста. Напротив, программист должен доказывать правильность программы одновременно с ее написанием. Третий довод существенно основан на следующем наблюдении. Если прежде всего задать себе вопрос, какова будет структура убедительного доказательства, и только затем строить программу, удовлетворяющую требованиям этого доказательства, то эта озабоченность правильностью оказывается очень эффективным эвристическим руководством. По определению этот подход применим только тогда, когда мы ограничиваемся программами с предсказуемым поведением, но он дает нам эффективное средство нахождения удовлетворительной программы среди множества таких программ.

Четвертый довод относится к способу, которым количество интеллектуальных усилий, необходимых для составления программы, зависит от длины программы. Высказывалось предположение, что существует некий закон природы, гласящий, что количество необходимых интеллектуальных усилий пропорционально квадрату длины программы. Однако, слава богу, никто так и не смог доказать этот закон. Причина в том, что он вовсе не обязательно справедлив. Все мы знаем, что единственное мыслительное средство, посредством которого вполне конечный фрагмент рассуждения может охватывать миллионы случаев, называется «абстракцией». Поэтому наиболее важным видом

деятельности компетентного программиста можно считать эффективную эксплуатацию его способности к абстрагированию. В этой связи может быть полезным подчеркнуть, что назначение абстракции не в том, чтобы быть неясной, но в том, чтобы создавать новый семантический уровень, в котором возможно достичь абсолютной точности. Конечно, я пытался отыскать фундаментальную причину, мешающую нашим механизмам абстрагирования быть достаточно эффективными. Но как бы я не старался, этой причины я найти не смог. Поэтому я склоняюсь к предположению, — до сих пор не опровергнутому опытом, — что при надлежащем применении наших способностей к абстракции интеллектуальное усилие, требующееся для написания или понимания программы, пропорционально не более чем длине самой программы. Побочный продукт этого исследования может иметь гораздо большее практическое значение, и действительно, он является основой для моего четвертого довода. Этим побочным продуктом было установление ряда способов абстрагирования, которые играют важную роль во всем процессе написания программ. Об этих абстрактных конструкциях известно столько, что можно было бы посвятить каждой из них целую лекцию. Какие последствия может иметь знание и сознательное использование этих приемов абстрагирования, открылось мне, когда я понял, что если бы это было известно 15 лет тому назад, то, например, шаг от БНФ к синтаксически ориентированным компиляторам занял бы несколько минут вместо нескольких лет. Следовательно, я выставляю наши новейшие знания о существенных способах абстрагирования в качестве четвертого довода.

Теперь обратимся к пятому доводу. Он имеет отношение к влиянию инструмента, который мы пытаемся использовать, на наши собственные мыслительные привычки. Я заметил некоторую культурную тенденцию, которая, по всей вероятности, своими корнями уходит в эпоху Возрождения, состоящую в том, чтобы не замечать этого влияния, считать человеческий ум высшим и независимым хозяином всего, что он порождает. Но если я начинаю анализировать свои собственные мыслительные привычки, а также и привычки своих коллег, то я независимо от своей воли прихожу совсем к другому выводу, а именно что инструменты, которые мы пытаемся использовать, а также язык или обозначения, применяемые нами для выражения или записи наших мыслей, являются главными факторами, определяющими нашу способность хоть что-то думать или выражать! Анализ влияния, которое имеют языки программирования на мыслительные привычки своих пользователей, и признание того, что теперь именно интеллектуальные ресурсы ограничивают наши возможности более чем что-либо другое, дают нам новый набор

критериев для сравнения относительных достоинств различных языков программирования. Компетентный программист вполне сознает жесткую ограниченность своих способностей; поэтому он подходит к задаче программирования в полном смиреннии и наряду с некоторыми другими вещами как чумы боится хитроумных трюков. Я слышал со всех сторон о хорошо известном диалоговом языке программирования, что, если программисты оснащены терминалом для него, происходит специфическое явление, известное под названием «однотрочники». Оно принимает одну из двух различных форм: один программист кладет перед другим однотрочную программу и либо гордо говорит, что она делает, и добавляет вопрос: «Можете закодировать это меньшим числом литер?» — как если бы это имело какое-нибудь значение! — или же просто говорит, «Угадайте, что она делает?» Из этого наблюдения мы должны заключить, что этот язык в качестве инструмента есть открытое приглашение к хитроумным трюкам; и хотя именно это может до некоторой степени объяснить его привлекательность для тех, кто любит демонстрировать свое интеллектуальное превосходство, но, прошу меня простить, я должен рассматривать это как одно из самых тяжких обвинений, которые можно выдвинуть против языка программирования. Другой урок, который мы должны извлечь из недавнего прошлого, — то, что разработка «более богатого» или «более мощного» языка программирования была ошибкой в том смысле, что эти барочные излишества, эти нагромождения вкусовых причуд действительно неудобоваримы и для компьютера, и для человека. Я предвижу большое будущее у очень систематических и очень скромных языков программирования. Когда я говорю «скромный», я имею в виду, что не только, например, от «оператора `for`» Алгола-60, но и от фортрановского «цикла `do`», быть может, придется отказаться как от слишком вычурных. Я провел несколько экспериментов с действительно опытными добровольцами-программистами, но произошло нечто совершенно незапланированное и неожиданное. Ни один из моих добровольцев не нашел очевидного и наиболее элегантно-го решения. При более пристальном изучении оказалось, что причина одна и та же: их понятие об итерации было так тесно связано с идеей дискретного приращения соответствующей управляемой переменной, что в их сознании существовала преграда, не позволяющая им видеть очевидное. Их решения были менее эффективными, неоправданно малопонятными, и их нахождение потребовало слишком много времени. Этот эксперимент меня поразил, но в то же время и многое объяснил. Наконец, в одном отношении есть надежда, что будущие языки программирования будут сильно отличаться от привычных ныне языков: в гораздо большей степени, чем до сих пор, они долж-



ны способствовать отражению в структуре того, что мы записываем, всех абстракций, необходимых для осознания сложности того, что мы разрабатываем. Вот и все о большей адекватности наших будущих инструментов, являющейся основой для пятого довода.

В качестве «реплики в сторону» я хочу добавить предупреждение тем, кто отождествляет сложность задачи программирования с борьбой против неадекватности наших современных инструментов, потому что они могут прийти к выводу, что, как только наши инструменты станут более адекватными, программирование перестанет быть проблемой. Программирование всегда останется очень трудным, поскольку, как только мы освободимся от вызванной несущественными обстоятельствами громоздкости, мы сразу же столкнемся с проблемами, которые пока далеко выходят за рамки того, что мы можем программировать.

Вы можете спорить с моим шестым доводом, поскольку не так легко собрать экспериментальные факты для его подтверждения, однако это не мешает мне верить в его справедливость. До сих пор я еще не упоминал слова «иерархия», но справедливо будет заметить, что это — ключевое понятие для всех систем, реализующих хорошо разбитое на части решение. Я могу даже пойти еще дальше и сделать из этого символ веры: мы в действительности можем решать удовлетворительным образом лишь такие задачи, которые в конечном итоге допускают хорошо разбивающееся на части решение. На первый взгляд эта картина человеческой ограниченности может поразить вас как крайне удручающая и безвыходная, но я так не думаю. Напротив, лучший способ научиться жить с нашей ограниченностью — это знать о ней. В настоящее время мы настолько скромны, что пытаемся находить только решения, разложимые на части, поскольку другие усилия ускользают от нашего понимания. Мы должны делать все возможное, чтобы избежать всех тех взаимодействий, которые подрывают нашу способность разложить систему на составные части полезным образом. И я не могу не ожидать, что это вновь и вновь будет приводить нас к открытию, что первоначально неподдающаяся разложению система в конце концов оказывается разложенной на составные части. Любой, кто видел, как большая часть неприятностей, возникающих на фазе компиляции, называемой «генерация выполняемого кода», может быть прослежена до странных свойств языка, на котором эта программа написана, узнает простой пример того, что я имею в виду. Широкая применимость хорошо разложимых на компоненты решений — это мой шестой и последний довод в пользу технической реализуемости революции, которая может произойти в ближайшие десять лет.

В принципе я предоставляю вам самим решать, сколь много веса вы придаете моим соображениям, поскольку вы прекрасно знаете, что я не могу никого заставить разделять мою веру. Как во всякой серьезной революции, я столкнусь с резкой оппозицией, и можно задать вопрос, откуда следует ожидать появления консервативных сил, пытающихся противодействовать такому ходу событий. Я не ожидаю, что они появятся главным образом из области большого бизнеса, и даже не из компьютерного бизнеса: скорее, я ожидаю их появления со стороны учебных заведений, которые в настоящее время занимаются подготовкой специалистов, а также со стороны пользователей компьютеров, которые находят свои старые программы столь важными, что не считают нужным переписывать или улучшать их. В этой связи, к сожалению, следует заметить, что во многих университетах выбор центральных вычислительных средств очень часто определялся требованиями немногих традиционных, но дорогостоящих приложений независимо от того, сколько тысяч «малых пользователей», желающих написать свои собственные программы, пострадают от этого выбора. Слишком часто, например, физика высоких энергий, как мне кажется, шапталжировала научную общественность стоимостью всего своего остального экспериментального оборудования. Самый легкий ответ, конечно,—это отрицание технической реализуемости, но я боюсь, что для этого вам понадобятся слишком сильные аргументы. Увы, нельзя рассчитывать на то, что интеллектуальный «потолок» среднего современного программиста станет препятствием для этой революции: когда другие начнут программировать намного эффективнее, тот, кто не сумел перестроиться, в любом случае окажется оттесненным в сторону.

Могут быть также политические препятствия. Даже если мы знаем, как обучать завтрашнего профессионального программиста, мы вовсе не уверены, что общество, в котором мы живем, позволит нам это сделать. Первым результатом от преподавания методологии — а не распространения знания — является увеличение способностей уже способных, тем самым увеличивается разброс интеллектуальных возможностей. В обществе, в котором система образования используется как инструмент для установления гомогенизированной культуры, в котором сливкам не дают подняться наверх, воспитание компетентных программистов должно быть политически неприемлемым.

Я хочу перейти к заключению. Автоматические компьютеры существуют вот уже четверть века. Они сильно повлияли на наше общество в качестве инструментов, но их воздействие в этом качестве не более чем рябь на поверхности нашей культуры по сравнению с более глубоким влиянием, которое они

окажут в качестве интеллектуального вызова, который не имеет прецедента в культурной истории человечества. По-видимому, иерархические системы обладают тем свойством, что некий объект, рассматриваемый как неделимое единство на некотором уровне, на более низком уровне (с большей степенью детализации) рассматривается как составной объект; в результате естественный масштаб пространства или времени, применимый на каждом уровне, уменьшается на порядок, когда мы переходим с одного уровня на следующий, более низкий. Мы понимаем, что такое стена, представляя ее как способ укладки кирпичей, кирпичи — как способ укладки кристаллов, кристаллы — как способ упаковки молекул и т. д. Поэтому число уровней иерархической системы, которые имеет смысл различать, примерно пропорционально логарифму отношения между самым крупным и самым мелким масштабом, и, следовательно, если только это отношение не слишком велико, мы не должны ожидать слишком большого числа уровней. В программировании на компьютере наш основной строительный «кирпич» имеет соответствующий временной масштаб порядка микросекунды, а наша программа может потребовать для своего выполнения нескольких часов вычислительного времени. Я не знаю другой технологии, охватывающей отношение в  $10^{10}$  раз или более: компьютер благодаря своему фантастическому быстродействию, по-видимому, первый среди устройств, обеспечивающих рабочую среду, в которой многоуровневая иерархия искусственных конструкций не только возможна, но и необходима. Этот вызов, т. е. столкновение с задачей программирования, является столь уникальным, что этот новый опыт может нам открыть глаза на нас самих. Он должен углубить наше понимание процессов проектирования и творчества; он должен научить нас лучше управлять задачей организации нашего мышления. Если он этого не сделает, то, на мой взгляд, мы вовсе недостойны иметь компьютер!

Он уже преподавал нам не один урок, и тот, который я хочу особо выделить, состоит в следующем. Мы будем лучше справляться с нашей работой программистов, если только мы будем подходить к этой работе с полным сознанием ее ужасающей сложности, если только мы будем верны скромным и элегантным языкам программирования, если мы будем учитывать природную ограниченность человеческого ума и приниматься за эту работу как Очень Смиренные Программисты.

## ЛИТЕРАТУРА

- [1] Some meditations on advanced programming, Proceedings of the IFIP Congress 1962, 535—538; Programming considered as a human activity, Proceedings of the IFIP Congress 1965, 213—217.
- [2] Solution of a problem in concurrent programming, control, CACM8 (Sept. 1965), 569; The structure of the «THE» multiprogramming system, CACM 11 (May 1968), 341—346.
- [3] Go to statement considered harmful, CACM 11 (Mar. 1968), 147—148.
- [4] A short introduction to the art of computer programming. Technische Hogeschool, Eindhoven, 1971.

## ПОСТСКРИПТУМ

Эдсгер Дейкстра  
Факультет информатики  
Университет штата Техас, Остин

Моя Тьюринговская лекция 1972 г. в значительной степени представляла собой изложение и обоснование принципиального убеждения: задача, стоящая перед программистом, — это интеллектуальный вызов высочайшего ранга. Меня поражает то обстоятельство, что и по сей день это кредо полностью сохранило свою актуальность: по-прежнему основная трудность информатики состоит в том, как не заблудиться в тех сложностях, которые создаем мы сами.

Однако высказанные в той лекции предложения, как следует отвечать на этот вызов, несомненно принадлежат времени, когда она была прочитана. Если бы я читал ее сегодня, то большую ее часть отвел бы роли формализованных методов программирования.

Сопоставление моих тогдашних ожиданий с тем, что действительно произошло за эти годы, рождает смешанные чувства. С одной стороны, мои самые дерзкие надежды исполнились с лихвой: ясные, четкие аргументы, ведущие к изощренным алгоритмам, которые было очень трудно или невозможно представить себе еще десять лет тому назад, стали постоянным источником интеллектуальных поисков. С другой стороны, меня огорчает, когда я вижу, сколь малая часть этих достижений вошла в обычные учебные курсы информатики, в которых эффективной разработкой высококачественных программ пренебрегают ради всяких модных поветрий (скажем, ради «постепенного самоулучшения дружелюбности интерфейсов экспертных систем»).

Есть какая-то верхняя грань скорости, с которой общество в состоянии усваивать прогресс, и, по-видимому, мне еще нужно учиться, как стать более терпеливым.

# программирование как искусство

*Дональд Е. Кнут*

Отрывок из постановления комитета по премиям Тьюринга 1974 г., процитированный его председателем Бернардом А. Галлером, на презентации этой лекции 11 ноября 1974 г. на ежегодной конференции АСМ в Сан-Диего.

Премия Тьюринга ежегодно присуждается ассоциацией АСМ человеку, внесшему вклад технического характера в деятельность программистского сообщества, причем такой вклад, который оказал существенное влияние на достаточно важную область информатики.

«Премия Тьюринга 1974 г. присуждается профессору Станфордского университета Дональду Е. Кнуту за ряд выдающихся работ в области анализа алгоритмов и разработки языков программирования и в особенности за его вклад в развитие «искусства программирования», осуществленный серией известных книг с этим общим названием. Методы программирования, алгоритмы, теоретические построения, изложенные в указанных книгах, послужили основой преподавания информатики и организующим началом в развитии этой дисциплины».

Однако эта официальная формулировка, обосновывающая присуждение премии Тьюринга за 1974 г., не может в полной мере отразить роль Дональда Кнута в современной информатике и компьютерной индустрии. Я знаком с профессором Аланом Дж. Перлманом, первым лауреатом премии Тьюринга, человеком, который обладает способностью в любой научной дискуссии настолько глубоко проникать в суть рассматриваемых проблем, что его высказывания неизменно оказываются в центре всего последующего обсуждения. Подобным образом алгоритмы, терминология, глубокие идеи, составившие содержание ряда блестящих книг и статей Дональда Кнута, также становятся предметом обсуждения почти во всех областях нашей науки. Такое дается нелегко. Всякому автору известно, какого тяжкого труда и величайшей организованности требует написание даже одного тома. Тем более не могут не вызывать восхищения ясность подхода, терпение и энергия, которыми должен был обладать Дональд Кнут, чтобы поставить перед собой задачу создать семь томов и с такой настойчивостью и скрупулезностью приступить к ее осуществлению.

Важно отметить, что эта премия и другие, которых он также был удостоен, присуждены ему после выхода трех томов его труда. Мы искренне рады признать заслуги и оценить преданность Дональда Кнута нашей науке. Мне чрезвычайно приятно быть председателем комитета, избравшего Дональда Кнута лауреатом премии Тьюринга Ассоциации АСМ 1974 г.

\* \* \*

Когда в 1959 г. начал публиковаться журнал *Communications of the ACM*, то члены редакционной коллегии АСМ, обсуждая задачи периодических изданий ассоциации, сделали следующее замечание [2]: «Если мы хотим, чтобы программирование стало важной частью информатики и компьютерных исследований, то следует всемерно способствовать превращению программирования из искусства в строгую науку». Эта же тема неоднократно поднималась и в последующие годы; так, в 1970 г. мы читаем о «первых шагах на пути превращения программирования из искусства в науку» [26]. Тем временем мы и в самом деле преуспели в преобразовании программирования в науку, причем замечательно простым способом — решив называть ее «компьютерной наукой»<sup>1)</sup>.

Во всех этих высказываниях неявно присутствует мысль о чем-то нежелательном для нас в той области человеческой деятельности, которую принято называть искусством; только став строгой Наукой, она может приобрести реальную значимость. Между тем вот уже 12 лет я работаю над серией книг под названием «Искусство программирования». Меня часто спрашивают, почему я выбрал такое название; причем некоторые, по-видимому, не верят, что я действительно назвал их именно так, поскольку мне пришлось видеть по крайней мере одну библиографическую ссылку на книги с названием «Дело программирования»<sup>2)</sup>.

В своей лекции я попытаюсь объяснить, почему я считаю подходящим именно слово «искусство»; я рассмотрю, что значит «быть искусством» и что значит в противоположность этому «быть наукой»; я попытаюсь выяснить, хороши ли искусства или плохи; и я попытаюсь показать, что правильный взгляд на предмет может помочь нам лучше делать свое дело.

Один из первых случаев, когда возник разговор о названии моих книг, произошел в 1966 г. во время последнего национального съезда АСМ в Южной Калифорнии. Это было еще до вы-

---

<sup>1)</sup> «computer science» — информатика.

<sup>2)</sup> В названии книг «The Art of Computer Programming» слово «art» («искусство») заменено на «act» («дело», «акт»).

хода в свет первого тома. Я припоминаю, что мы с моим другом ужинали в отеле, и он, зная, насколько высоким было мое самомнение уже в те годы, спросил, правда ли, что я собираюсь назвать свои книги «Введением в Дона Кнута». Я ответил, что, напротив, собираюсь назвать их как раз в *его* честь. Звали его Арт Эванс<sup>1)</sup>. («Искусство программирования собственной персоной».)

Из этой истории мы можем заключить, что слово «art» («искусство») многозначно. Пожалуй, самое замечательное в этом слове — то, что оно употребляется в нескольких различных смыслах, и каждый из них вполне применим к программированию. Готовясь к лекции, я отправился в библиотеку, чтобы выяснить, что писали люди о слове «искусство» в разные времена. Проведя несколько дней за этим увлекательнейшим занятием, я пришел к выводу, что слово «art», вероятно, одно из самых интересных слов в английском языке.

## ИСКУССТВА В ДРЕВНОСТИ

Если обратиться к латинским истокам, то мы обнаружим слова «ars», «artis», означающим «умение». Примечательно также, что соответствующее греческое слово *τεχνη* является общим корнем слов «техника» и «технология».

В наше время, когда говорят об «искусстве», то это слово, вероятно, ассоциируется в первую очередь с «изящными искусствами», такими, как живопись или скульптура, но до XX в. в это слово вкладывался совсем другой смысл. Поскольку прежнее значение слова «искусство» все еще проявляется во многих идиомах, особенно когда искусство противопоставляется науке, то я хотел бы потратить несколько минут на рассуждения о слове «искусство» в его классическом смысле.

Первые университеты, основанные в средние века, предназначались для обучения семи так называемым «свободным искусствам»<sup>2)</sup>, а именно грамматике, риторике, логике, арифметике, геометрии, музыке и астрономии. Заметим, что набор дисциплин весьма далек от программы современных гуманитарных колледжей и что по крайней мере три из семи свободных искусств являются важными компонентами информатики. В те времена под искусством понималось нечто, порождаемое силой человеческого интеллекта, в отличие от видов деятельности, вытекающих из природных потребностей или инстинктов. «Свободные искусства» — это освобожденная, духовная деятель-

<sup>1)</sup> Art Evans, art (англ.) — искусство.

<sup>2)</sup> «Liberal Arts» — в современном английском языке это словосочетание означает «гуманитарные науки».

ность в противоположность искусствам, связанным с физическим трудом, вроде пахоты (ср. с [6]). В средние века слово «искусство» само по себе обычно обозначало логику [4], под которой в свою очередь понималось изучение силлогизмов.

## НАУКА В ПРОТИВОПОЛОЖНОСТЬ ИСКУССТВУ

Слово «наука», по-видимому, долгое время употреблялось примерно в том же смысле, что и «искусство», говорили, например, также о семи свободных науках, тех же самых, что и семь свободных искусств [1]. Дунс Скотус в XIII в. называл логику «Наукой Наук, Искусством Искусств» (см. 12, с. 34f). С развитием цивилизации и познания слова эти приобретали все более самостоятельные значения: под наукой понималось знание, под искусством — приложение знания. Так, наука астрономия составляла фундамент искусства навигации. То есть различие между этими словами было примерно такое же, как в наше время между словами «наука» и «техника».

В XIX в. о соотношении между наукой и искусством писали многие авторы, и, на мой взгляд, наилучшее рассуждение по этому поводу принадлежит Джону Стюарту Миллю. В его работе [28], 1843 г., говорится, в частности, следующее:

Для того чтобы составить фундамент одного искусства, чаще всего необходимо бывает несколько наук. Человеческая деятельность слишком сложна, поэтому для того, чтобы что-то сделать, зачастую требуется знать природу и свойства многих вещей... Искусство, вообще говоря, состоит из ряда научных истин, организованных наиболее удобным образом не с точки зрения их логического осмысления, а с точки зрения их практического применения. Наука организует и упорядочивает свои истины таким образом, чтобы как можно шире охватить с единых позиций закономерности Вселенной. Искусство... соединяет в себе многие сведения из области науки, иногда далекие друг от друга, относящиеся к созданию разнообразных и разнообразных условий, необходимых для произведения всевозможных действий, вызванных потребностями практической жизни.

Просматривая многие и многие рассуждения о смысле слова «искусство», я обнаружил, что уже по крайней мере в течение двух столетий различные авторы призывают к превращению искусства в науки. Например, в предисловии к учебнику минералогии, написанному в 1784 г., читаем: «До 1780 г. минералогия, хотя и воспринималась многими с некоторой натяжкой как Искусство, едва ли могла считаться Наукой».

В большинстве словарей слово «наука» определяется как логически организованное и систематизированное знание в форме общих «законов». Преимущество науки заключается в том, что она избавляет нас от необходимости обдумывать множество частных случаев, позволяя мыслить с помощью понятий более высокого уровня абстракции. Как писал в 1853 г. Джон



Раскин [32]: «Задача науки состоит в том, чтобы видимость заменять фактами, впечатления — доказательствами».

Я думаю, что если бы авторы, работы которых я изучал, жили в наши дни, то они согласились бы с такой формулировкой: «Наука — это знание, настолько понятное нам, что мы могли бы обучить ему вычислительную машину; там, где мы еще не все до конца понимаем, начинается область искусства». Понятие алгоритма или программы — чрезвычайно полезный тест глубины наших познаний о каком-то предмете, поэтому превращение искусства в науку означает, что мы способны автоматизировать данную область деятельности.

Несмотря на то что искусственный интеллект достиг значительного прогресса, существует тем не менее огромная пропасть между тем, что способны делать обычные люди, и тем, что смогут в обозримом будущем делать вычислительные машины. Таинственные вспышки озарения, проявляющиеся у людей, когда они говорят, слушают, занимаются творчеством и даже пишут программы, все еще находится вне досягаемости науки; поэтому почти все, что мы делаем, до сих пор является искусством.

С этой точки зрения, разумеется, весьма желательно превращение программирования в науку, и за 15 лет, прошедшие со дня публикации тех замечаний, которые я процитировал в начале своей лекции, мы несомненно проделали немалый путь в этом направлении. Пятнадцать лет назад мы настолько плохо понимали, что такое программирование, что даже *помыслить* не могли о доказательствах правильности программ; мы просто-напросто гоняли программу до тех пор, пока не «убеждались», что она работает. В те времена мы не представляли даже, как достаточно строго сформулировать саму *идею* правильности программы. Лишь в последние годы мы начали разбираться в процессах абстракции, лежащих в основе конструирования и понимания программ. Наше новое знание о программировании дает громадную практическую отдачу, несмотря на то что лишь для небольшого числа программ реально получено строгое доказательство правильности; однако благодаря ему мы стали лучше понимать принципы построения программ. Суть в том, что теперь, когда мы пишем программу, мы уже знаем, что в принципе при желании могли бы построить строгое доказательство ее правильности, так как нам уже известно, как формулируются подобные доказательства. Этот новый научный фундамент позволяет писать программы, значительно более надежные, чем те, что мы писали раньше, когда в основе суждений об их правильности лежала только интуиция.

Область «автоматизации программирования» — одно из основных направлений исследований по искусственному интеллек-

ту. Приверженцы этого направления были бы счастливы, если бы могли прочесть лекцию под названием «Программирование как артефакт<sup>1)</sup>» (имея в виду, что программирование превратилось всего-навсего в реликвию минувших времен), поскольку их цель — создать машину, способную по одной только спецификации задачи написать программу лучше любого из нас. Лично я не думаю, что эта цель когда-либо может быть вполне достигнута, тем не менее я убежден, что эти исследования имеют чрезвычайно важное значение, поскольку любое новое знание о программировании помогает нам усовершенствоваться в своем искусстве. В таком смысле следует постоянно стремиться превращать в науку всякое искусство, поскольку этот процесс способствует развитию самого искусства.

Я не могу устоять перед искушением рассказать здесь еще один случай, имеющий отношение к науке и искусству. Несколько лет назад, когда я был в Чикагском университете, при входе в одно из зданий я обратил внимание на две таблички. На одной из них было написано «Информатика» и изображена стрелка направо. На другой было написано «Информация» и нарисована стрелка налево. Иначе говоря, наука об информации идет одним путем, искусство информации — другим.

## НАУКА И ИСКУССТВО

Итак, мы пришли к выводу, что современное программирование является одновременно и искусством, и наукой и что эти два аспекта замечательнейшим образом друг друга дополняют. По-видимому, и большинство других авторов, задавшихся аналогичным вопросом, приходили точно к такому же выводу — что их предмет также является и наукой, и искусством независимо от того, о каком именно предмете идет речь (см. [25]). Я обнаружил самоучитель по фотографии, написанный в 1893 г., в котором утверждается, что «занятие фотографированием — это одновременно и искусство, и наука» [13]. На самом деле, даже взяв в руки словарь, для того чтобы справиться о значениях слов «искусство» и «наука», и случайно заглянув в предисловие редактора, я обнаружил, что оно начинается словами: «Составление словаря есть одновременно наука и искусство». Редактор словаря Funk & Wagnall [27] замечает, что кропотливый процесс накопления и классификации сведений о словах носит научный характер, в то время как тщательное формулирование их определений требует навыков точного и

---

<sup>1)</sup> Артефакт — в археологии так называют любой обнаруженный при раскопках предмет, сделанный человеческими руками, в отличие от находок, имеющих естественное происхождение. — *Прим. ред.*

лаконичного изложения. «Наука без искусства вряд ли будет полезной, искусство без науки заведомо было бы неточным».

Готовясь к этой лекции, я изучил картотеку Станфордской библиотеки, для того чтобы посмотреть, как другие авторы употребляют слова «искусство» и «наука» в названиях своих книг. Это оказалось небезынтересным занятием.

Я обнаружил, например, две книги с названием «Искусство игры на фортепиано» [5, 15], а также книги с названиями «Научный подход к технике игры на фортепиано» [10] и «Наука практической игры на фортепиано» [30]. Нашлась также книга под названием «Искусство игры на фортепиано: научный подход» [22].

Затем мне попалась небольшая приятная книжица под названием «Нежное искусство математики» [31]. Тут я слегка взгрустнул оттого, что, пожалуй, не смог бы вполне искренне назвать программирование «нежным искусством».

Несколько лет назад мне довелось познакомиться с книгой под названием «Искусство вычислений», выпущенной в Сан-Франциско в 1879 г. автором по имени С. Фрушер Говард [14]. Это была книга по практической деловой арифметике, распроданная в количестве свыше 400 000 экземпляров в нескольких изданиях, вышедших до 1890 г. Меня весьма развлекло чтение предисловия к этой книге, так как из него видно, что взгляды Фрушера и смысл, вкладываемый им в заголовок книги, в корне отличаются от того, что имел в виду я; он пишет: «Знание Науки о числах совершенно не имеет значения, навыки в Искусстве вычислений абсолютно необходимы».

В заголовках некоторых книг употребляются оба слова, «искусство» и «наука», например: «Наука бытия и искусство жития» Махариши Махеш Йоги [24]. Существует также книга под названием «Искусство научного открытия» [11], в которой анализируется, как были сделаны некоторые великие научные открытия.

Однако довольно о классическом понимании слова «искусство». На самом деле, выбирая название для своих книг, я думал вовсе не об искусстве в таком значении, а прежде всего о теперешнем содержании этого слова. Пожалуй, одна из интереснейших книг, с которыми мне довелось познакомиться в ходе моих исследований, — это недавняя работа Роберта М. Мюллера «Наука искусства» [29]. Из всех уже упоминавшихся изданий книга Мюллера ближе всего подходит к выражению той идеи, которую я хотел бы сделать основной темой своей сегодняшней лекции, рассматривая истинное содержание искусства, как мы его понимаем сегодня. Он замечает: «Когда-то считалось, что образное мышление, присущее людям искусства, смертельно для науки. И наоборот, сухая логика науки, полагали

тогда, звучит как смертный приговор для любых взлетов фантазии». Далее он исследует преимущества синтеза науки и искусства.

Научный подход обычно характеризуют такими словами, как логический, систематический, холодный, беспристрастный, рациональный, в то время как художественный подход характеризуют словами эстетический, творческий, гуманитарный, эмоциональный, иррациональный. Мне представляется, что применительно к программированию оба этих на первый взгляд несовместимых подхода чрезвычайно ценны.

Эмма Лехмер в 1956 г. писала, что она считает программирование «в одинаковой мере и точной наукой, и захватывающим искусством» [23]. Х. С. М. Коксетер в 1957 г. заметил, что иногда чувствует себя «не столько ученым, сколько художником» [7]. Именно в это время Ч. П. Сноу высказывает свою тревогу по поводу углубляющейся пропасти между «двумя культурами» среди образованных людей [34, 35]. Он указывает, что если мы действительно стремимся достичь прогресса, то должны научиться сочетать ценности науки и искусства.

## ПРОИЗВЕДЕНИЯ ИСКУССТВА

Когда я сижу в аудитории, слушая длинную лекцию, то примерно к этому времени внимание мое обычно начинает рассеиваться. Поэтому я хотел бы знать: не слишком ли вас утомили мои разглагольствования об «искусстве» и «науке»? Я все же надеюсь, что вы в состоянии внимательно дослушать остаток лекции, поскольку как раз сейчас мы подошли к той ее части, которая мне особенно дорога.

Когда я говорю о программировании как искусстве, то я думаю о нем в первую очередь как о некоторой художественной *форме* в эстетическом смысле. Главную цель своей деятельности как преподавателя и автора я вижу в том, чтобы помочь людям научиться писать *красивые программы*. Именно поэтому мне было особенно приятно узнать [33], что мои книги даже появились в библиотеке Изящных искусств Корнуэльского университета. (Однако мои три тома, по-видимому, прочно осели на полках без всякого употребления; боюсь, что служащие библиотеки ошиблись, истолковав название моих книг слишком буквально.)

Я чувствую, что составление программ сродни сочинению стихов или музыки. Как сказал Андрей Ершов [9], программирование способно дать нам одновременно и интеллектуальность, и эмоциональное удовлетворение, поскольку овладеть сложностью и установить систему согласованных правил — это настоящий подвиг.

Более того, читая программы, написанные другими людьми, мы можем воспринимать некоторые из них как настоящие произведения искусства. Я до сих пор не могу забыть волнения, охватившего меня, когда я в 1958 г. читал листинг ассемблерной программы SOAP II Стэна Поля. Возможно, вы сочтете меня сумасшедшим, да и вкусы с тех пор переменились, но в то время для меня было чрезвычайно важно увидеть, насколько изящной может быть системная программа, в особенности по сравнению с теми тяжеловесными кодами, которые я обнаружил, изучая листинги других программ того времени. Возможность писать красивые программы даже на ассемблере — именно это в первую очередь сделало меня приверженцем программирования.

Бывают программы изящные, щегольские, бывают блестящие программы. Я убежден, что можно создавать *грандиозные* программы, *благородные* программы и даже поистине *великие* программы!

Недавно я обсуждал эти идеи с Микаэлем Фишером, и он высказал мысль о том, что программисты могли бы продавать свои авторские программы коллекционерам как произведения искусства. АСМ могла бы основать специальную экспертную комиссию для подтверждения подлинности каждого вновь поступившего программистского изделия, а квалифицированные торговые агенты и люди особой новой профессии — критики программ — назначили бы цену. Осуществив эту идею, мы получили бы неплохой способ повысить свои доходы.

## ВКУС И СТИЛЬ

Если говорить серьезно, то я рад, что идея стиля в программировании наконец-то обрела признание, и надеюсь, что большинство из вас уже видели великолепную маленькую книжку Кернинга и Плуджера «Начала стилистики программирования» [16]. В этой связи важно помнить о том, что не существует единственного «наилучшего» стиля; каждый из нас имеет свои предпочтения, и было бы ошибкой пытаться принуждать людей к каким-то неестественным для них формам. Часто приходится слышать высказывания: «Я не разбираюсь в искусстве, но я знаю, что мне нравится». Важно, чтобы вам в самом деле нравился стиль, которого вы придерживаетесь; он должен служить для вас наилучшим способом самовыражения.

Эдсгер Дейкстра подчеркивал этот момент в предисловии к своей книге «Краткое введение в искусство программирования» [8].

Моя цель состоит лишь в том, чтобы объяснить важность хорошего стиля в программировании, однако представленные в книге конкретные сти-

листические элементы служат лишь для иллюстрации тех преимуществ, которые вообще можно получить, используя понятие «стиль». В этом отношении я чувствую себя сродни преподавателю композиции в консерватории. Он не учит своих подопечных тому, как следует писать конкретную симфонию, но он должен помочь им обрести собственный стиль и объяснить им, что под этим подразумевается. (Эта аналогия и привела меня к мысли говорить именно об «Искусстве программирования».)

Зададимся теперь вопросом: что же такое хороший стиль? что такое плохой стиль? В этом отношении не следует быть слишком суровым в суждениях о работах других людей. Философ Джереми Бентам в начале девятнадцатого века сформулировал эту мысль так [3, книга 3, гл. 1]:

Те, кто судят об изяществе и хорошем вкусе, мнят себя благодетелями рода человеческого, в то время как на самом деле они лишь мешают людям получать наслаждение... Нет такого вкуса, который безусловно заслуживал бы эпитета *хороший*, если только это не вкус к занятиям, которые помимо производимой ими приятности заключают в себе и некоторую долю полезности; не существует и такого вкуса, который можно было бы безусловно охарактеризовать как плохой, если только это не вкус к занятиям, имеющим вредную направленность.

Настаивая на своих предубеждениях и пытаясь «реформировать» вкус другого человека, мы, возможно, сами того не осознавая, лишаем его вполне законного удовольствия. Поэтому я не берусь осуждать многое из того, что делают другие программисты, даже если ни за что не стал бы делать так, как они. Важно лишь, что *сами они* воспринимают создаваемое ими как нечто прекрасное.

В отрывке, который я только что процитировал, Бентам дает нам одно указание относительно того, какие из принципов эстетики предпочтительнее прочих, а именно «полезность» результата. Мы можем располагать определенной свободой в выборе собственных критериев прекрасного, однако замечательно, если изделие, которое кажется нам красивым, воспринимается другими как полезное. Я должен признаться, что обожаю писать программы, и больше всего я люблю писать программы, наилучшие в каком-либо отношении.

Существует, разумеется, множество точек зрения, согласно которым программа может оцениваться как «хорошая». Во-первых, очень хорошо, если программа правильно работает. Во-вторых, часто бывает желательно, чтобы программу можно было легко изменять, когда возникает такая необходимость. Обе эти цели достигаются, если программа легкочитаема и понятна для человека, знакомого с соответствующим языком.

Другое важное свойство, которым должна обладать хорошая программа, — корректное взаимодействие с пользователем, в особенности это касается обработки ошибок во входных данных. Придумывание осмысленных сообщений об ошибках, раз-

работка гибких форматов входных данных, не провоцирующих пользователя на ошибки, есть настоящее искусство.

Еще один важный аспект качества программ — эффективность использования ресурсов вычислительной машины. К сожалению, многие сейчас пренебрегают эффективностью, говоря, что это дурной вкус. Причина заключается в том, что мы переживаем реакцию на те времена, когда эффективность считалась единственным достойным внимания критерием качества программ и программисты были настолько озабочены эффективностью, что ради ее достижения излишне усложняли свои программы. В результате чрезмерных ухищрений эффективность фактически сводилась на нет из-за сложности отладки и сопровождения программ. Суть в том, что программисты тратили слишком много усилий, заботясь об оптимизации не там и не тогда, когда нужно. Поспешная, непродуманная оптимизация является корнем всех (или почти всех) зол в программировании.

Грош была бы нам цена, если бы наши представления об оптимизации сводились лишь к процентам потерянного или выигранного пространства или времени счета. Когда мы покупаем автомобиль, большинству из нас безразлична разница в цене в 50 или 100 долларов. В то же время мы готовы предпринять специальную поездку в определенный магазин, чтобы приобрести за 25 центов вещь, которая стоит 50 центов. Я думаю, что всему свое время и свое место — и заботам об эффективности в том числе. Свои взгляды по поводу истинной роли эффективности я изложил в статье о структурном программировании, которая появится в очередном номере журнала *Computer Surveys* [21].

## МЕНЬШЕ СРЕДСТВ — БОЛЬШЕ УДОВОЛЬСТВИЯ

Я обратил внимание на одно любопытное обстоятельство, связанное с эстетическим удовлетворением, — мы получаем значительно больше удовольствия, если нам удастся добиться чего-либо ограниченными средствами. Например, программа, которой я более всего доволен и горд, — это компилятор, который я написал когда-то для примитивной мини-машины с памятью всего лишь в 4096 слов по 16 бит в слове. Человек, сумевший достичь результата при столь суровых ограничениях, может чувствовать себя поистине виртуозом.

Аналогичные явления наблюдаются и в других сферах жизни. Например, люди частенько влюбляются в свои фольксвагены, но мало кто способен влюбиться в линкольн континенталь (хотя он ездит гораздо лучше). В те времена, когда я изучал

программирование, популярным развлечением было составление программ, которые бы помещались на одной перфокарте и выполняли бы как можно больше действий. Думаю, что примерно такое же удовольствие испытывают приверженцы АПЛ, сочиняя свои «однострочники». Любопытно, что и теперь, когда мы преподаем программирование, редко удается по-настоящему увлечь студента, до тех пор пока он не пройдет курс, включающий непосредственную практику на мини-машинах. Работа на мощных машинах с изощренными операционными системами и языками, видимо, не способствует зарождению теплых чувств к программированию, по крайней мере вначале.

Не совсем ясно, как применить этот принцип, для того чтобы сделать работу программистов более привлекательной. Ясно, что программисты не обрадовались бы, если бы начальство вдруг объявило, что будут установлены новые машины с вдвое меньшей памятью, чем прежние. Думаю, вряд ли можно надеяться, что даже самые преданные «асы» программирования будут рады такому новшеству, потому что никто не любит без необходимости расставаться с имеющимися удобствами. Приведу другой пример, который поможет прояснить дело. В 20-х годах режиссеры отчаянно сопротивлялись введению звукового кино, поскольку они законно гордились своим умением передавать человеческую речь без звука. Аналогично, истинные мастера программирования вполне могли бы воспротивиться введению более мощных аппаратных средств; современные устройства массовой памяти, похоже, губят всю прелесть старых добрых ленточных методов сортировки. Однако современные режиссеры вовсе не собираются возвращаться назад к немому кино, и не потому что ленивы, а потому что знают, что и с более совершенной техникой можно делать великолепные картины. Форма искусства изменилась, но остался неограниченный простор для мастерства.

Каким же образом оттачивают они свое умение? Похоже, лучшие режиссеры разных времен осваивали свое искусство в относительно примитивных условиях, часто в других странах, не имеющих развитой киноиндустрии. И наиболее интересные результаты в программировании также, похоже, были получены людьми, не имеющими доступа к большим машинам. Мораль этих наблюдений, по-моему, заключается в том, что мы должны сознательно использовать в образовании идею ограниченных ресурсов. Каждый из нас лишь выиграет, создавая время от времени «игрушечные» программы с заданными искусственными ограничениями, заставляющими нас до предела напрягать свои способности. Нельзя все время купаться в роскоши — это делает нас апатичными. Искусство решения мини-задач на пределе своих возможностей оттачивает наше умение для ре-



альных задач; к тому же, имея такой опыт, мы получим больше удовольствия, работая в менее ограниченных условиях.

Точно так же не нужно шарахаться от «искусства для искусства» или испытывать неловкость по поводу программ, написанных просто для развлечения. Когда-то я получил массу удовольствия, написав программу на Алголе, которая состояла из одной инструкции обращения к процедуре вычисления скалярного произведения, но таким необычным способом, что вместо вычисления этой величины она вычисляла *m*-е простое число [19]. Несколько лет назад студенты Станфорда были увлечены задачей нахождения кратчайшей программы на Фортране, которая бы печатала сама себя, в том смысле, что текст ее выдачи был идентичен ее собственному исходному тексту. Такую же задачу можно рассмотреть и для других языков. Не думаю, что это было для них пустой тратой времени; да и Джереми Бентам, которого я цитировал выше, вряд ли стал бы отрицать полезность подобного времяпрепровождения [3, кн. 3, гл. 1]. «Напротив, — писал он, — не существует занятия, полезность которого была бы более неоспоримой. В чем же заключается тогда смысл полезности, как не в том, чтобы служить для нас источником удовольствия?»

## О ПОЛЬЗЕ ХОРОШИХ ИНСТРУМЕНТОВ

Одна из особенностей современного искусства — подчеркнута творческий характер. Многие художники в наше время, похоже, менее всего озабочены созданием прекрасных произведений; важна лишь новизна идеи. Я бы не советовал уподоблять программирование современному искусству в таком смысле, но это наводит на одну мысль, которая представляется мне важной. Бывает, что приходится выполнять почти безнадежно-скучное программистское задание, которое не оставляет практически никаких возможностей для проявления творчества. В такой ситуации некто мог бы прийти ко мне и сказать: «И так-то ваше распрекрасное программирование? Хорошо вам рассуждать о том, что я, мол, должен получать удовлетворение, создавая очаровательные изящные программы, но как, скажите на милость, я могу превратить в искусство эту нудную работу?»

Что же, это правда, не каждая программистская работа сама по себе может доставить удовольствие. Возьмите домохозяйку, которой каждый день приходится вытирать один и тот же стол; не всякая ситуация дает простор для проявления творчества и мастерства. Но и в этом случае дело можно поправить: даже рутинная работа может доставлять удовольствие, если предметы, с которыми вы имеете дело, красивы. На-

пример, вытирание изо дня в день одного и того же стола может и в самом деле нравиться, если это великолепно отделанный стол, изготовленный из дерева ценной породы.

Бывают ситуации, когда мы призваны не сочинять, а *исполнять* симфонию; но исполнение прекрасного музыкального произведения — истинное наслаждение, несмотря на то что наша свобода ограничена предписаниями композитора. Так и программист иногда выступает в роли не столько художника, сколько ремесленника, но и работа ремесленника может доставлять радость, если он имеет дело с хорошими инструментами и материалами.

Поэтому в заключение я хотел бы обратиться к системным программистам и конструкторам вычислительных машин, создающим системы, которыми пользуются все остальные: «Пожалуйста, дайте нам такие инструменты, которыми было бы приятно пользоваться, а не такие, с которыми приходится вести постоянную борьбу. Пожалуйста, дайте нам инструменты, которые улучшали бы наше настроение и тем вдохновляли бы на создание более качественных программ».

Мне бывает очень трудно убедить своих молодых коллег в том, что программирование прекрасно, поскольку первое, что я должен им объяснять, — это как пробивается перфокарта «слэш слэш JOB равно то-то и то-то»<sup>1)</sup>. Даже язык управления заданиями можно сконструировать таким образом, чтобы с ним было приятно иметь дело, и он вовсе не обязан быть чисто функциональным.

Конструкторы вычислительной техники могли бы сделать свои машины более приятными в употреблении, например предоставив плавающую арифметику, удовлетворяющую простым математическим законам. То, что мы имеем сейчас, делает чрезвычайно трудным строгий численный анализ; должным образом организованные операции могли бы стимулировать специалистов численного анализа на создание более качественных подпрограмм с гарантированной точностью (см. [20, с. 204]).

Посмотрим теперь, что могут сделать разработчики программного обеспечения. Лучшее средство для поддержания хорошего расположения духа у пользователя — это предоставить ему программы, с которыми можно взаимодействовать. Не нужно делать системы слишком уж автоматизированными, так что вся их работа проходит где-то за сценой, — следует оставлять и программисту-пользователю какие-то возможности для проявления творческих возможностей. Общая черта, свойственная всем программистам, — любовь к непосредственной работе за машиной; так давайте же не будем лишать их этого удовольствия.

<sup>1)</sup> //JOB = ... — одна из команд языка управления заданиями.

ствия. С какими-то заданиями лучше справляется машина, другие лучше предоставить человеческому разумению; правильно построенная программа должна обеспечивать верное соотношение между тем и другим. (В течение многих лет я стремился избегать излишней автоматизации, см. [18].)

Хорошим примером здесь могут служить системы для изменения программ. Многие годы программисты не имели понятия о реальном распределении вычислительных затрат по отдельным участкам программ. Опыт показывает, что почти все программисты неправильно представляют себе, где на самом деле находятся узкие места в их программах; неудивительно, что попытки оптимизации так часто оказывались бесплодными, если программисты не имели информации о реальном распределении времени счета по строкам программы. Подобные усилия напоминают попытки молодой пары составить рациональный семейный бюджет, не имея понятия о ценах на продукты, одежду и домашнюю утварь. Единственное, что предоставлялось программисту, — это оптимизирующий компилятор, который проделывает нечто таинственное над транслируемой программой, но никогда не объясняет, что же он делает. К счастью, наконец-то забрезжила надежда на появление систем, которые все же предполагают наличие у пользователя кое-какого интеллекта; они автоматически предоставляют инструментовку программы и соответствующие указания о реальном распределении затрат. Эти экспериментальные системы имели громадный успех потому, что они дают измеримое улучшение качества программ, и еще потому, что пользоваться ими — одно удовольствие. Поэтому я убежден, что такие системы обязательно станут стандартным элементом программного обеспечения — это лишь вопрос времени. В моей статье в журнале *Computer Surveys* [21] можно найти дальнейшее обсуждение этого вопроса, а также некоторые соображения по поводу того, как интерактивные программы могут способствовать более полному удовлетворению программистов-пользователей.

Долг разработчиков языков — создать такие языки программирования, которые способствовали бы хорошему стилю, поскольку известно, что стиль существенно зависит от языка, на котором формулируется программа. Всплеск интереса к структурному программированию, который мы сейчас переживаем, показал, что ни один из существующих языков не является идеальным инструментом для манипулирования с программами и структурами данных; неясно также, каким должен быть идеальный язык. Поэтому я предполагаю в ближайшие годы провести ряд тщательных экспериментов в области конструирования языков.

## ЗАКЛЮЧЕНИЕ

Итак, мы видели, что программирование — это искусство, поскольку оно является приложением накопленных знаний для практических целей, поскольку оно требует умения и мастерства, и в особенности потому, что продукты программирования могут представлять эстетическую ценность. Программист, который бессознательно ощущает себя художником, получает удовольствие от своей работы и справляется с ней лучше. Поэтому мы можем только радоваться тому, что люди, выступающие на конференциях по программированию, говорят о «State of the Art».

## ЛИТЕРАТУРА

1. Bailey, Nathan. The Universal Etymological English Dictionary. T. Cos. London, 1787. См. «Art», «Liberal» и «Science».
2. Bauer, Walter F., Juncosa, Mario L., and Perlis, Alan J. ACM publication policies and plans. J. ACM 6 (Apr. 1959), 121—122.
3. Bentham, Jeremy. The Rationale of Reward. Trans. from Theorie des peines et des recompenses, 1811, by Richard Smith, J. & H. L. Hunt, London, 1825.
4. The Century Dictionary and Cyclopedia 1. The Century Co., New York, 1889.
5. Clementi, Muzio. The Art of Playing the Piano. Trans. from L'art de jouer le pianoforte by Max Vogrich. Schirmer, New York, 1898.
6. Colvin Sidney. «Art» Encyclopaedia Britannica, eds. 9, 11, 12, 13, 1875—1926.
7. Coxeter, H. S. M. Convocation adress, Proc. 4th Canadian Math. Congress, 1957, pp. 8—10.
8. Dijkstra, Edsger W. EWD316: A Short Introduction to the Art of Programming. T. H. Eindhoven, The Netherlands, Aug. 1971.
9. Ershov, A. P. Aethetics and human factors in programming. Comm. ACM 15: [July 1972], 501—505.
10. Fielden, Thomas. The Science of Pianoforte Technique. Macmillan, London, 1927.
11. Gore, George. The Art of Scientific Discovery. Longmans, Green, London, 1878.
12. Hamilton, William. Lectures on Logic 1. Wm. Blackwood, Edinburgh, 1874.
13. Hodges, John A. Elementary Photography: «The Amateur Photographer» Library 7. London, 1893. Sixth ed., revised and enlarged, 1907, p. 58.
14. Hovard C. Frusher. Hovard's Art of Computation and golden rule for equation of payments for schools, business colleges and selfculture... C. F. Hovard, San Francisco, 1879.
15. Hummel, J. N. The Art of Playing the Piano Forte. Boosey, London, 1827.
16. Kernighan B. W., and Plauger, P. J. The Elements of Programming Style. McGraw-Hill, New York, 1974.
17. Kirwan, Richard. Elements of Mineralogy. Elmsy, London, 1784.
18. Knuth, Donald E. Minimizing drum latency time. J. ACM 8 (Apr. 1961), 119—150.
19. Knuth, Donald E. and Merner, J. N. ALGOL 60 confidential. Comm. ACM 4: (June 1961), 268—272.
20. Knuth, Donald E. Seminumerical Algorithms: The Art of Computer Programming 2. Addison — Wesley, Reading, Mass., 1969.
21. Knuth, Donald E. Structured programming with go to statements. Computing Surveys 6 (Dec. 1974), 261—301.

22. Kochevitsky, George. *The Art of Piano Playing: A Scientific Approach*. Summary — Birchard, Evanston, Ill., 1967.
23. Lehmer, Emma. Number theory on SWAC. *Prom. Symp. Applied Math. Soc.* (1956), 103—108.
24. Mahesh Yogi, Maharishi. *The Science of Being and Art of Living*. Allen & Unwin, London, 1963.
25. Malevinsky, Moses L. *The Science of Playwriting*. Brentano's, New York, 1925.
26. Manna, Zohar, and Pnueli, Amir. Formalization of properties of functional programs. *J. ACM* 17 (July 1970), 555—569.
27. Markwardt, Albert H. Preface to Funk and Wagnall's *Standard College Dictionary*. Harcourt, Brace & World, New York, 1963, vii.
28. Mill, John Stuart. *A System of Logic, Ratiocinative and Inductive*. London, 1843. Цитаты взяты из предисловия, § 2, и из Книги 6, гл. 11 (12 в более поздних изданиях), § 5.
29. Mueller, Robert E. *The Science of Art*. Jhon Day, New York, 1967.
30. Parsons, Albert Ross. *The Science of Pianoforte Practice*. Schirmer, New York, 1886.
31. Pedoe, Daniel. *The Gentle Art of Mathematics*. English U. Press, London, 1953.
32. Ruskin, Jhon. *The Stones of Venice* 3. London, 1853.
33. Salton, G. A. Частная переписка. Июнь 21, 1974.
34. Show, C. P. The two cultures. *The New Statesman and Nation* 52 (Oct. 6, 1956), 413—414.
35. Snow, C. P. *The Two Cultures: and a Second Look*. Cambridge University Press, 1964.

# Логика и языки программирования

Д. С. Скотт

Оксфордский университет

[Д. С. Скотт является одним из двух лауреатов премии Тьюринга 1976 г., врученной ему на ежегодной конференции АСМ 20 октября в Хьюстоне. Статья другого лауреата, М. О. Рабина, «Сложность вычислений» напечатана на стр. 371.]

Уже долгое время логика пытается выяснить, являются ли ответы на те или иные вопросы принципиально вычислимыми, поскольку решение этих проблем устанавливает границы возможностей формализации. Последние достижения теории сложности позволили получать точные оценки эффективности методов решений. Однако эти методы применимы только к логике, а значимость логических методов в других, более прикладных областях теории вычислений остается пока неясной.

Языки программирования, обладая весьма развитой синтаксической формализацией, очевидно, более всего подходят для выяснения этого вопроса, однако семантическая теория языков еще далека от завершенности. Несмотря на множество примеров, мы пока не можем дать достаточно общие, математически обоснованные ответы на следующие вопросы: что такое вычислительная машина? что такое процесс вычисления? каким образом (или насколько хорошо) вычислительная машина моделирует процесс? Программы, естественно, входят в описание процессов. Поэтому определение точного смысла некой программы требует объяснения того, что является объектами вычисления (в некотором смысле статику проблемы) и каким образом они трансформируются (динамику проблемы).

До сих пор теории автоматов и сетей, хотя и наиболее интересные с точки зрения динамики, формализовали только часть области, и похоже, что внимание уделялось в основном алгебраическим аспектам и конечным наборам состояний. По-видимому, понимание свойств более сложных программ вовлекает нас в область бесконечных объектов и заставляет использовать несколько уровней объяснения на пути от концептуальных идей к окончательному моделированию на реальных ЭВМ. Эти уровни могут быть математически точно обоснованы, если найти правильные абстракции для представления необходимых структур.

Многие независимые исследователи использовали метод представления типов данных в качестве решеток (или методов частичной упорядоченности), упорядоченных относительно содержания информации, и их непрерывные отображения. Эти работы продемонстрировали гибкость данного подхода в обеспечении определений и доказательств, прозрачных и относительно независимых от реализации. Тем не менее необходимо сделать еще очень много, чтобы показать, каким образом абстрактная концептуализация может (или не может) быть реализована, прежде чем заявить, что мы имеем единую теорию.

Как последнему солауреату премии Тьюринга, мне доставляет величайшее удовольствие разделить ее и эту трибуну с

Микаэлем Рабином. К сожалению, со времени написания нашей статьи в 1959 г. мы почти не имели возможности для сотрудничества, что является для меня большой потерей: при совместной работе я добиваюсь больших результатов. Но организовать подходящие условия для такого сотрудничества нелегко, особенно на стыке научных дисциплин, когда людей к тому же разделяют государственные границы. Тем не менее я с большим интересом и восхищением следил за его работой. Как вы сегодня уже слышали, Рабину удалось *применить* идеи логики, связанные с разрешимостью, вычислимостью и сложностью, к вопросам, представляющим действительный интерес в математике и теории вычислений. Он, как и многие другие, активно работает над созданием новых методов анализа для широкого и весьма перспективного класса алгоритмических проблем. Эти аспекты теории вычислений находятся вне моей компетенции, так как за прошедшие годы мои научные интересы и интересы Рабина разошлись. Начиная с конца 1960-х годов я сосредоточился в основном на возможности применения логических идей для лучшего *концептуального* понимания языков программирования. Поэтому я больше не буду детально говорить о нашей совместной работе, а остановлюсь на своих собственных разработках, планах и надеждах.

Трудности получения точного и всеобъемлющего представления о языке возникли в период разработки громоздких «универсальных» машинных языков. В настоящее время, как мне кажется, мы стоим на пороге еще одной технологической революции, которая полностью перевернет наше представление о вычислительных машинах и их программном обеспечении. (Я только что отметил стремление АСМ полностью исключить термин «машина».) Возможно, что громоздкие языки окажутся недостаточно гибкими, однако, я думаю, проблема *семантики* наверняка останется. Хотелось бы верить, что мои исследования, проведенные опять-таки совместно с другими учеными, и в особенности с покойным Кристофером Стречи, явились существенным вкладом в основание работ по семантике. Поживем — увидим. Надеюсь также, работы по семантике недолго будут оставаться несвязанными с исследованиями в области, разрабатываемой Рабином.

## ПОПЫТКА ОПРАВДАТЬСЯ НЕ ОПРАВДЫВАЯСЬ

Вообще говоря, я считаю, что выступающие не должны оправдываться: это только вызывает у аудитории чувство неловкости. Но на такой встрече, как эта, одно оправдание все же необходимо (вместе с отречением).

Те из вас, кому известна моя профессиональная подготовка, могут вспомнить сэра Николаса Гимкрэка, героя пьесы «Виртуоз», написанной Томасом Шэдуэллом в 1676 г. с целью слегка пошутить над замечательными опытами, проводившимися в то время Лондонским Королевским обществом. В одной из сцен пьесы сэр Николас лежит на столе и пытается научиться плавать, имитируя движения лягушки в чаше с водой. На вопрос, пробовал ли *он* научиться плавать *в воде*, сэр Николас отвечает, что ненавидит воду и никогда не подойдет к ней! «Мне достаточно теоретического изучения процесса плавания, — отвечает он, — мне безразлична его практическая польза. Я редко довожу что-либо до практического применения... Знание есть моя конечная цель».

Несмотря на совпадение моих и сэра Гимкрэка конечных целей, мне хотелось бы отмежеваться от пренебрежения практическими аспектами. Дело, однако, в том, что у меня отсутствует практический опыт в современном программировании, и по необходимости мне пришлось ограничиться наблюдением программирования, получая знания «из вторых рук», наблюдая действия «лягушек и других существ». К счастью, некоторые из этих «лягушек» умеют говорить, и мне пришлось изучать с ними чужой для меня язык. Однако вполне возможно, что я не до конца понимал, о чем идет речь. Но я пытался читать и быть в курсе происходящего. Прошу простить мой непрофессионализм в области программирования, и я, естественно, не хотел бы выступать с поучениями: большинство выступавших были в этом смысле более подготовлены и сообщили нам много ценного. Единственное, что я пытался сделать, — это довести некоторые результаты из области логики, которые, как мне казалось, могут быть полезны в программировании, до тех, кто мог бы их использовать. Я также пытался получить свои собственные результаты, и вам судить, насколько мне это удалось.

К моему большому удовлетворению, сегодня мне не нужно извиняться за недостаток опубликованного материала; мне нужно было бы это сделать, если бы я написал свою речь в день получения приглашения. В августе была напечатана замечательная статья Роберта Теннета [14] по денотационной семантике, и я горячо рекомендую ее как отправную работу по этой теме. Теннет не только дает серьезные примеры, идущие намного дальше опубликованного мной и Стречи, но и представляет хорошо продуманную библиографию.

В прошлом месяце вышла значительная книга Милна и Стречи. К несчастью, совершенно неожиданная и безвременная кончина Стречи не позволила ему приступить к пересмотру и исправлению рукописи. С его смертью мы потеряли много в стиле и проницательности (не говоря уже о вдохновении), однако



Роберт Милн замечательно довел до конца их совместную работу. Самым значительным в этой книге является то, что проблема сложного языка обсуждается в ней от *начала до конца*. Кому-то изложение может показаться слишком строгим, однако суть в том, что семантика в этой работе — не плод произвольных догадок, а реально обоснована. Она является результатом серьезного и глубокого осмысления, и у нас есть возможность детально разобраться и решить, может ли данный подход быть плодотворным. Милн построил описание таким образом, что стало возможным восприятие языка на многих уровнях, вплоть до конечного компилятора. Он не пытался обойти трудности. Хотя эта книга и не написана тем язвительным и беспечным языком, каким обычно пользовался сам Стречи, тем не менее ее можно считать достойным памятником заключительному этапу работы Стречи, к тому же она содержит немало оригинальных идей Милна. (Я могу так говорить потому, что сам не участвовал в написании этой книги.)

Недавно была опубликована не очень большая, но интересная работа Донахью [4], в которой содержатся не обсуждавшиеся ранее или обсуждавшиеся, но с другой точки зрения, вопросы. Она написана совершенно независимо от меня и Стречи, и я был очень рад ее появлению.

Скоро выйдет учебник Джо Стоя [13], который дополнит работы предыдущих авторов и, видимо, явится хорошим учебным пособием, так как Стой имеет немалый опыт преподавательской работы как в Оксфордском университете, так и в Массачусетском технологическом институте.

Из фундаментальных работ сейчас должна появиться моя собственная переработанная статья [12]. Она была написана с точки зрения использования операторов перечисления в более «классической» теории рекурсии, и ее связь с практическим программированием на первый взгляд может показаться не совсем очевидной. Но меня успокаивает то, что другие работы объясняют использование этой теории именно так, как я и предполагал.

К счастью, все вышеперечисленные авторы широко цитируют литературу, и поэтому сегодня нет необходимости вдаваться в исторические детали. Я позволю себе только сказать, что и многие другие исследователи использовали различные идеи Стречи, как и мои идеи, и упоминание об их работах можно найти не только в библиографиях, но и, например, в последних докладах Мана [7] и Бёма [1]. Пусть меня простят те, кого я невольно не назвал сейчас, — они знают, как я ценю их содействие и интерес к моим работам.

## НЕМНОГО О СЕБЕ

Я родился в Калифорнии и приступил к работе в области математической логики, будучи студентом младших курсов в Беркли в начале 1950-х годов. Больше всего на меня повлияли, конечно, Альфред Тарский, его коллеги и студенты Калифорнийского университета. Наряду с другими предметами под руководством Рафаэля и Джулии Робинсон, которых я хочу поблагодарить за многие ценные идеи, я изучал теорию рекурсивных функций. В то же время самостоятельно я познакомился с  $\lambda$ -исчислением Чёрча и Карри (которое в начале в буквальном смысле было для меня кошмаром). Особенно большое влияние на меня оказало изучение семантики Тарского и его определение истинности для формализованных языков. Эти концепции, как вы знаете, до сих пор широко обсуждаются в философии естественного языка. Я пытался применить идеи подхода Тарского к алгоритмическим языкам, преимущество которых заключается по крайней мере в том, что они достаточно хорошо синтаксически формализованы. Возможно, требует обсуждения, действительно ли мне удалось, руководствуясь схемами Стречи и других ученых, найти правильные определения терминов. Именно я первым заявил о том, что *не все* проблемы решаются одним лишь подбором денотатов к некоторым языкам: для языков типа (чистого)  $\lambda$ -исчисления нужные определения в большинстве случаев найдены, однако многие понятия программирования до сих пор остаются неопределенными.

Свою дипломную работу я закончил в Принстоне в 1958 г. Руководителем моим был Алонсо Чёрч, который в то же время руководил диссертацией Микаэля Рабина. Тогда мы и встретились с Рабином, а нашу совместную работу по теории автоматов сделали в 1957 г. во время летней практики в ИВМ. Конечно, мы не были единственными в этой области, однако нам удалось разработать несколько основных идей. В то время я уже думал о проекте математического определения вычислительной машины. Сейчас мне кажется, что метод конечных состояний представляет лишь частичный интерес и не имеет особого практического значения. Правда, большинство физических вычислительных машин можно смоделировать как устройства с конечным числом состояний, однако эта *ограниченность* (конечность) едва ли является их наиболее значительной чертой, и взгляд на ЭВМ как на автомат часто бывает весьма поверхностным.

Два последних достижения в области теории автоматов кажутся мне наиболее интересными, по крайней мере с точки зрения математики: многоуровневая иерархия Хомского и связь с полугруппами. С точки зрения алгебры (во всяком случае,

по моему мнению) Эйленберг, этот Евклид теории автоматов, в своих работах [5] действительно сказал последнее слово. Я хочу подчеркнуть и то, что он сумел обойтись без теории абстрактных категорий. Категории могут привести к хорошим решениям (см. Мэйнз [7]), но слишком раннее их использование значительно затрудняет понимание. Это мое личное мнение.

В некоторых аспектах иерархия Хомского в конце концов вызывает разочарование. Контекстно-свободные языки действительно очень важны, и всем необходимо ознакомиться с ними. Однако мне не совсем ясно, что за этим следует, если вообще что-либо следует. Существует множество других семейств языков, но из хаоса не возникло так уж много порядка. Я не думаю, что в этой области уже сказано последнее слово. Не зная истинного направления и разочаровавшись в теории автоматов, казавшейся мне чрезмерно сложной, я оставил работу в этой области. Однажды я попытался связать автоматы и языки программирования, предложив более систематический способ отделения машины от программы. Эйленбергу эта идея совершенно не понравилась, но мне было очень приятно увидеть в последней книге Кларка и Коуэлла [2] ее элегантно воплощенное, предложенное Питером Ландином. Признаю, что это не алгебра, но мне кажется, что это не что иное, как (элементарное, хотя и в некоторой степени теоретическое) программирование. Хотелось бы увидеть следующий шаг, который должен привести к чему-то между Манном [8] и Милном — Стречи [9].

В Принстоне я впервые познакомился с настоящей вычислительной машиной, «доисторической» машиной фон Неймана. И благодарить за это я должен Эктона Формана. Эта машина кажется сейчас совсем устаревшей, однако она была *действующей*, и работа на ней доставила мне и Хейлу Троттеру большое удовлетворение. Как печально было видеть ее мертвый корпус в Смитсонианском музее без каких-либо признаков того, какой она была при жизни.

Из Принстона я перешел в Чикагский университет, где в течение двух лет преподавал на математическом факультете. Там я и встретился с Бобом Эшенхёрстом и Ником Метрополисом. К сожалению, мое короткое пребывание в университете не позволило мне поработать с ними: как обычно, факультеты слишком отдалены друг от друга. (Естественно, что я говорю только о связях с программированием и не пытаюсь говорить о своей работе в области математики и логики.)

Затем в течение трех лет я работал в Беркли, где через Гарри Хаски и Рене де Вожелера познакомился со многими специалистами в области программирования. Последний также ознакомил меня в деталях с языком Алгол-60. В то время в Беркли еще не было факультета информатики. Через некоторое

время по личным причинам мне пришлось переехать в Станфорд. Таким образом, хотя я и преподавал теорию вычислений в течение одного семестра, моя работа в Беркли не имела особого значения. Единственное, о чем я всегда буду сожалеть, — это то, что я не изучил досконально работу Дика и Эммы Лехмер, чей способ получения *результатов* в теории чисел с помощью ЭВМ восхищает меня до сих пор. Теперь, когда с помощью ЭВМ решена проблема четырех красок, мы становимся свидетелями развития методов автоматического доказательства теорем. Мне очень жаль, что я не занимался этой проблемой.

Все согласятся с тем, что с начала 1960-х годов факультет информатики в Станфорде был одним из лучших в стране, и, конечно, непонятно, почему я все-таки ушел оттуда. Возможно, дело в том, что моя должность объединяла работу на факультетах философии и математики. Мне кажется, что мой личный недостаток заключается в том, что я не всегда знаю, где мне следует быть и чем я хочу заниматься. Однако оставим личные переживания. У меня сложились хорошие отношения с аспирантами на факультете в Форсайте, где мы провели замечательные лекции и семинары. Лично на меня и на мои представления в области теории вычисления оказали большое влияние Джон Маккарти и Пэт Саппес, а также остальные члены их группы. Вместе с Солом Феферманом и Георгом Крайзелем мы создали сильную группу логиков. В эту группу аспирантов-логиков входил и Ричард Платек, работа которого произвела на меня особое впечатление несколько лет спустя, когда я понял способ применения некоторых из его идей.

В это же время я получил годичный отпуск для поездки в Амстердам, который неожиданно оказался поворотным пунктом в моем творческом развитии. Я не буду углубляться в детали, так как это сложная история, отмечу только, что 1968/69 академический год стал для меня годом глубокого кризиса, и мне до сих пор больно вспоминать об этом. К счастью или к несчастью, Пэт Саппес предложил включить меня в рабочую группу 2.2 Международной федерации по обработке информации (в настоящее время — группа формального описания концепций программирования). Председателем ее был тогда Том Стил, и на заседании в Вене я впервые встретился с Кристофером Стречи. Дискуссии, возникавшие в этой группе, носили настолько острый характер, что я был действительно рад тому, что не принимал участия в работе комитетов по более значительным вопросам, как, например, комитета по Алголу. Тем не менее я полагаю, что дискуссии являются отличным терапевтическим средством: они выявляют лучшие и худшие качества людей. Во всяком случае, нужно научиться защищать себя. Из всех участников этих баталий я выделил Стречи, его стиль, его

мысли мне более всего импонировали, хотя, как мне показалось, он иногда сильно преувеличивал значение некоторых своих идей. Однако некоторые из них побудили меня заняться более глубоким изучением проблемы.

Лишь в конце моего годичного пребывания в Амстердаме я приступил к обсуждению некоторых проблем с Жако де Баккером, и только в переписке, которую мы вели в течение всего лета, удалось окончательно оформить наши идеи. Значительное влияние на меня оказала и Венская группа IBM, с деятельностью которой я ознакомился, участвуя в рабочей группе 2.2. К этому времени я решил перейти на факультет философии Принстонского университета, но прежде, чем возвратиться домой, я попросил дополнительный отпуск и осенью 1969 г. заехал к Стречи в Оксфорд. Этот период стал для меня периодом наиболее активной деятельности: в течение многих дней у меня было что-то вроде нервной горячки. Сотрудничество со Стречи на протяжении этих нескольких недель стало лучшим временем в моей профессиональной жизни. Наша новая встреча состоялась летом следующего года в Принстоне. К сожалению, к 1972 г., когда я окончательно перебрался в Оксфорд, мы оба были настолько заняты преподавательской работой и административными проблемами, что наше сотрудничество оказалось практически невозможным. К тому же Стречи был настолько обескуражен постоянным отсутствием средств на исследовательские работы и помощи в преподавательской деятельности, что в конце концов он ушел из университета и занялся вместе с Милном подготовкой к публикации своей книги. (Для него это было перенапряжением и отрицательно сказалось на здоровье. Как много бы я дал, чтобы он увидел свою работу опубликованной!)

Возвращаясь к 1969 г., должен сказать, что я начал с того, что показал Стречи его *полную неправоту* и необходимость делать все совершенно иначе. В самом начале Роджер Пенроуз привлек его внимание к  $\lambda$ -исчислению, и он разработал удобный способ использования этой нотации функциональной абстракции в объяснении концепций программирования. Тем не менее этот способ был *формальным*, и я пытался доказать, что у него не было математической базы. Я уже рассказывал эту историю и, чтобы не быть многословным, скажу только, что мне удалось убедить его отказаться от бестипового  $\lambda$ -исчисления. Однако вскоре, по мере того как одно следствие из моего предложения следовало за другим, я стал понимать, что вычислимые функции могут быть определены в большом разнообразии пространств. Настоящим шагом вперед стало понимание того, что функциональные пространства действительно являются хорошими пространствами, и я ясно помню, как логик Анджей

Мостовски, который в это же время находился в Оксфорде, просто не мог поверить, что тип функциональных пространств, который я определил, вообще мог иметь конструктивное описание. Но когда я сам в этом убедился, у меня родилась мысль о существовании более удивительных, чем мы могли предположить, возможностях использования функциональных пространств. При появлении сомнений в полезности принудительной жесткости логических типов, в чем я пытался убедить Стречи, я пришел к выводу, что одно из этих пространств изоморфно своему собственному функциональному пространству, которое обеспечивает модель «бестипового»  $\lambda$ -исчисления. Окончание этой истории описано в литературе.

[Интересные сведения, проливающие свет на историю  $\lambda$ -исчисления, дает участие в ней Алана Тьюринга. Он учился в Принстоне вместе с Чёрчем и связал вычислимость с (формальным)  $\lambda$ -исчислением в 1936—1937 гг. У Кроссли можно найти пояснения об отношении Стива Клини к его работе (и к последующему влиянию  $\lambda$ -исчисления) (Кроссли [3]). (Конечно, поздние взгляды Тьюринга на компьютеры оказали на Стречи большое влияние, однако сейчас не время проводить полный исторический анализ.) Хотя я никогда не встречался с Тьюрингом (он умер в 1954 г.), соприкосновение с ним через Чёрча, Стречи и моих нынешних коллег по Оксфорду, Ле Фокса и Робина Ганди, оказалось достаточно близким несмотря на то, что в то время я только заканчивал университет, Чёрч больше не занимался  $\lambda$ -исчислением и мы никогда не обсуждали его работу с Тьюрингом.]

Я нахожу очень странным то обстоятельство, что мои модели  $\lambda$ -исчисления не были найдены кем-либо ранее, но я очень воодушевлен тем, что новые типы моделей с новыми свойствами, как, например, домены Гордона Плоткина [10], открываются сейчас. Лично я уверен, что для этого все уже подготовлено как на теоретическом, так и на прикладном уровнях. Джон Рейнолдс и Роберт Милн независимо друг от друга ввели новый индуктивный метод доказательства эквивалентностей; Робин Милнер продолжает в Эдинбурге интересную работу по LCF и технике доказательств. Начало направлению доказательства через модели было положено теоремой Дэвида Парка о связи оператора неподвижной точки с так называемым парадоксальным комбинатором  $\lambda$ -исчисления, и это положило начало изучению бесконечных, но вычисляемых операторов, которое продолжается по многим линиям. Другое направление разрабатывается в Новосибирске под руководством Ю. Л. Ершова, а Карл Х. Хофманн и его группа ознакомили меня с очень интересными связями с топологической алгеброй. К сожалению, я не могу здесь перечислить всех, работающих в этой области.

Мне особенно приятно доложить этому собранию, что Тони Хоар недавно согласился принять кафедру вычислений в Оксфордском университете, которая освободилась после того, как умер Стречи. Это назначение открывает новые возможности для сотрудничества как с Хоаром, так и с теми студентами, которых он привлечет в свою группу, как только займет этот пост в будущем году. Как вы понимаете, практическим аспектам использования и создания компьютерных языков и методологии программирования в Оксфорде будет придаваться особое значение (что делал и Стречи), и это действительно пойдет на пользу всем; я надеюсь, что открываются и новые прекрасные возможности для теоретических исследований.

### НЕКОТОРЫЕ СЕМАНТИЧЕСКИЕ СТРУКТУРЫ

Обращаясь к технической стороне, я хотел бы кратко пояснить сущность действия моей конструкции и возможности ее развития. Здесь мы не будем доказывать «правильность» этих абстракций, ее обоснование можно найти в упомянутых ранее работах.

Проще всего показать сущность этой конструкции на двух областях:  $\mathcal{B}$  — области булевских значений, и  $\mathcal{P} = \mathcal{B}^\infty$  — области бесконечных последовательностей булевских значений. Важным моментом является то, что мы собираемся принять идею *частичных* функций, при математическом представлении которых функции время от времени принимают *частичные значения*. Использование  $\mathcal{B}$  делает эту идею очень простой. Запишем:

$$\mathcal{B} = \{\text{истина, ложь, } \perp\},$$

где  $\perp$  является дополнительным элементом, называемым «неопределенность». Для того чтобы закрепить за  $\perp$  его место, введем на области  $\mathcal{B}$  частичный порядок  $\sqsubseteq$ , где

$$x \sqsubseteq y \text{ тогда и только тогда, когда или } x = \perp, \text{ или } x = y,$$

для всех  $x, y$  из  $\mathcal{B}$ . Мы можем читать « $\sqsubseteq$ » как то, что информация, содержащаяся в  $x$ , содержится в информации, содержащейся в  $y$ . Элемент  $\perp$  поэтому содержит пустую информацию. Эта схема показана на рис. 1.

(Примечание: во многих статьях я отстаивал использование решеток, которые как частичные порядки имеют и «дно» («нижний» элемент)  $\perp$ , и «верх» («верхний» элемент)  $\top$ , что позволяет ут-

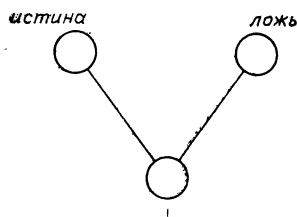


Рис. 1. Булевские значения.

верждать, что  $\perp \leq x \leq \top$  верно для всех элементов области. Это предположение не было безоговорочно принято по многим причинам, о которых я не буду говорить. Некоторые рассуждения о разумности этого можно найти у Скотта [12], хотя, конечно, исследуемые им структуры очень специфичны. Вероятно, лучше всего не включать, но и не исключать  $\top$ , и для простоты я не буду более упоминать об этом сегодня.)

Рассмотрим теперь  $\mathcal{P}$ , область последовательностей. Мы будем использовать нотацию, где нижние индексы обозначают координаты; таким образом, для всех  $x \in \mathcal{P}$

$$x = \langle x_n \rangle_{n=0}^{\infty}.$$

Каждый терм таков, что  $x_n \in \mathcal{B}$ , потому что  $\mathcal{P} = \mathcal{B}^{\infty}$ . Здесь имеется в виду «прямое произведение», и формально мы определяем частичный порядок  $\leq$  на  $\mathcal{P}$  следующим образом:

$x \leq y$  тогда и только тогда, когда  $x_n \leq y_n$  для всех  $n$ .

Неформально это означает, что последовательность  $y$  «лучше» информационно, чем последовательность  $x$ , тогда и только тогда, когда некоторые координаты  $x$ , которые были «неопределенными», становятся «определенными» при переходе от  $x$  к  $y$ . Например, каждая из следующих последовательностей находится в отношении  $\leq$  к следующей за ней:

$\langle \perp, \perp, \perp, \perp, \dots \rangle,$

$\langle \text{истина}, \perp, \perp, \perp, \dots \rangle,$

$\langle \text{истина}, \text{ложь}, \perp, \perp, \dots \rangle,$

$\langle \text{истина}, \text{ложь}, \text{истина}, \perp, \dots \rangle.$

Очевидно, что этот список может быть бесконечно расширен, и не обязательно рассматривать координаты в строгом порядке  $n=0, 1, 2, \dots$ . Таким образом, отношение  $\leq$  на  $\mathcal{P}$  является более сложным, чем определенное ранее на  $\mathcal{B}$ .

Очевидным различием между  $\mathcal{B}$  и  $\mathcal{P}$  является то, что  $\mathcal{B}$  конечно, в то время как  $\mathcal{P}$  имеет бесконечно много элементов. В  $\mathcal{P}$  также некоторые элементы имеют бесконечное информационное содержание, тогда как в  $\mathcal{B}$  это не так. Однако мы можем использовать частичный порядок на  $\mathcal{P}$  для абстрактного объяснения того, что мы подразумеваем под «конечной аппроксимацией» и «пределами». Последовательности, представленные выше, являются конечными в  $\mathcal{P}$ , так как они имеют только конечное число координат, отличных от  $\perp$ . Любой данный  $x \in \mathcal{P}$  можно обрезать до конечного элемента, определяя

$$(x \upharpoonright m)_n = \begin{cases} x_n, & \text{если } n < m; \\ \perp, & \text{если это не так.} \end{cases}$$



Из наших определений видно, что

$$x \uparrow m \subseteq x \uparrow (m+1) \subseteq x,$$

так что  $x \uparrow m$  «стремятся» к пределу; и фактически этот предел есть первоначальный  $x$ . Запишем это следующим образом:

$$x = \bigcup_{m=0}^{\infty} (x \uparrow m),$$

где  $\bigcup$  есть  $\sup$  или операция вычисления наименьшей верхней грани в частично упорядоченном множестве  $\mathcal{P}$ . Дело в том, что  $\mathcal{P}$  имеет много  $\sup$ ; и для любых элементов  $y^{(m)} \subseteq y^{(m+1)}$  в  $\mathcal{P}$  (независимо от того, конечны они или нет) мы можем определить предел  $z$ , где

$$z = \bigcup_{m=0}^{\infty} y^{(m)}.$$

(Совет: спросите себя, какие координаты будут у  $z$ .) Мы не можем пересказать здесь все детали, но  $\mathcal{P}$  действительно является топологическим пространством и  $z$  действительно является пределом. Поэтому, хотя  $\mathcal{P}$  и бесконечно, мы можем вернуться в область *конечных* операций и обсуждать *вычислимые* операции на  $\mathcal{P}$  и на более сложных областях.

Кроме последовательностей и частично упорядоченных структур на  $\mathcal{P}$  мы можем определить много видов алгебраических структур. Именно поэтому  $\mathcal{P}$  является подходящим примером. Например, с точностью до изоморфизма пространство  $\mathcal{P}$  удовлетворяет

$$\mathcal{P} = \mathcal{P} \times \mathcal{P},$$

где в правой части равенства подразумевается обычное бинарное прямое произведение. Область  $\mathcal{P} \times \mathcal{P}$  состоит из всех упорядоченных пар  $\langle x, y \rangle$ , где  $x, y \in \mathcal{P}$ , отношение  $\subseteq$  определяется на  $\mathcal{P} \times \mathcal{P}$  как

$$\langle x, y \rangle \subseteq \langle x', y' \rangle \text{ тогда и только тогда, когда } x \subseteq x' \text{ и } y \subseteq y'.$$

Но для всех практических целей можно отождествить  $\langle x, y \rangle$  с последовательностью, уже содержащейся в  $\mathcal{P}$ . Действительно, покоординатно мы можем определить

$$\langle x, y \rangle_n = \begin{cases} x_k, & \text{если } n = 2k; \\ y_k, & \text{если } n = 2k + 1. \end{cases}$$

Введенный выше критерий для  $\subseteq$  между парами будет выполнен, и мы можем сказать, что на  $\mathcal{P}$  определена парная функция (отображающая два значения в одно).

Парная функция  $\langle ., . \rangle$  на  $\mathcal{P}$  обладает многими интересными свойствами. В сущности, мы уже заметили, что она *монотонна* (как только увеличивается информационное содержание  $x$  и  $y$ , увеличивается информация, содержащаяся в  $\langle x, y \rangle$ ). Более важно, что функция  $\langle ., . \rangle$  является *непрерывной* в следующем определенном смысле:

$$\langle x, y \rangle = \bigcup_{m=0}^{\infty} \langle x \uparrow m, y \uparrow m \rangle,$$

что означает, что функция  $\langle ., . \rangle$  хорошо ведет себя при конечных аппроксимациях. И это только один пример. Для данного подхода важна целая *теория* монотонных и непрерывных функций.

Используя даже такую небольшую структуру, которую мы определили на  $\mathcal{P}$ , можно дать описание *языка*. Для иллюстрации остановимся на двух изоморфизмах, которым удовлетворяет  $\mathcal{P}$ , а именно  $\mathcal{P} = \mathcal{B} \times \mathcal{P}$  и  $\mathcal{P} = \mathcal{P} \times \mathcal{P}$ . Первый определяет  $\mathcal{P}$  как нечто связанное с (бесконечными) последовательностями булевских значений, тогда как второй напоминает о приведенной выше парной функции. На рис. 2 приводится краткое БНФ-определение (использующее форму Бэкуса — Наура) языка с двумя видами выражений: булевское значение ( $\beta$ ) и последовательность булевских значений ( $\sigma$ ).

```

 $\beta ::= \text{true} \mid \text{false} \mid \text{head } \sigma$ 
 $\sigma ::= \beta^* \mid \beta \sigma \mid \text{tail } \sigma \mid$ 
         if  $\beta$  then  $\sigma'$  else  $\sigma'' \mid$ 
         even  $\sigma \mid \text{odd } \sigma \mid \text{merge } \sigma' \sigma''$ 

```

Рис. 2. Краткий язык.

Этот язык действительно очень краткий: без переменных, без объявлений, без присваиваний, только небольшой набор константных термов. Заметим, что выбранная нотация предназначена для того, чтобы сделать значение этих выражений очевидным. Так, если  $\sigma$  обозначает последовательность  $x$ , то **голова**  $\sigma$  (**head**) должна обозначать первый терм  $x_0$  в последовательности  $x$ . Так как  $x_0 \in \mathcal{B}$  и  $x \in \mathcal{P}$ , то мы сохраняем смысл введенных нами типов объектов.

В более строгом смысле для каждого выражения мы сможем определить его (константное) значение  $[[\cdot]]$ ; так что  $[[\beta]] \in \mathcal{B}$  для булевских выражений  $\beta$  и  $[[\sigma]] \in \mathcal{P}$  для выражений над последовательностями. Так как БНФ-определение языка содержит десять видов предложений, мы могли бы выписать десять равенств, чтобы полностью определить семантику приведенного языка; мы удовольствуемся здесь некоторыми из

них. Принимая во внимание замечания предыдущего абзаца, запишем:

$$[\text{head } \sigma] = [\sigma]_0.$$

Выражение  $\beta^*$  создает бесконечную последовательность булевских значений:

$$[\beta^*] = \langle [\beta], [\beta], [\beta], [\beta], \dots \rangle.$$

(Эта нотация, хотя и приближительная, но понятная.) Точно так же определим

$$[\beta\sigma] = \langle [\beta], [\sigma]_0, [\sigma]_1, [\sigma]_2, \dots \rangle;$$

и

$$[\text{tail } \sigma] = \langle [\sigma]_1, [\sigma]_2, [\sigma]_3, [\sigma]_4, \dots \rangle.$$

Далее

$$[\text{even } \sigma] = \langle [\sigma]_0, [\sigma]_2, [\sigma]_4, [\sigma]_6, \dots \rangle;$$

и

$$[\text{merge } \sigma' \sigma''] = \langle [\sigma'], [\sigma''] \rangle$$

Этого достаточно, чтобы дать представление о самой идее. Но нужно понять, что здесь мы обсуждаем одно из возможных представлений, что  $\mathcal{P}$  удовлетворяет значительно большему числу изоморфизмов (например,  $\mathcal{P} = \mathcal{P} \times \mathcal{P} \times \mathcal{P}$ ) и существует очень много способов — причем вычислимых — разбивать на части и комбинировать последовательности булевских значений.

## ФУНКЦИОНАЛЬНОЕ ПРОСТРАНСТВО

Неверно было бы считать, что вся моя идея исчерпывается только содержанием изложенного. Это соответствовало бы элементарному уровню программных схем (см. ван Эмден — Ковальски [6] или Манна [8], последняя глава). То, что некоторые называют «семантикой неподвижной точки» (мне самому термин «fixpoint» не нравится), является лишь *первой* частью моей работы. Вторая же часть включает функции, в качестве аргументов которых выступают также функции, т. е. функции

высших порядков, и тем самым мы выходим за уровень программных схем. В самом деле, методы неподвижной точки могут быть применены и к функциям высшего порядка, но это не единственное их достоинство. Для более полного определения в качестве семантической структуры необходимо использовать *функциональное пространство*. Я пытался обратить на это особое внимание еще в 1969 г., но многие тогда не понимали меня.

Предположим, что  $\mathcal{D}'$  и  $\mathcal{D}''$  — два домена обсуждаемого нами вида (скажем,  $\mathcal{R}$ , или  $\mathcal{R} \times \mathcal{R}$ , или  $\mathcal{S}$ , или что-либо еще). Под  $[\mathcal{D}' \rightarrow \mathcal{D}'']$  будем понимать домен *всех* монотонных и непрерывных функций  $f$ , отображающих  $\mathcal{D}'$  в  $\mathcal{D}''$ . Именно это я и имею в виду под термином «функциональное пространство». Это не сложно с математической точки зрения, но и не совсем очевидно, что  $[\mathcal{D}' \rightarrow \mathcal{D}'']$  опять-таки является доменом того же самого вида, хотя, возможно, и более сложной структуры. Я не могу этого здесь доказывать, но по крайней мере я могу определить отношение  $\equiv$  на функциональном пространстве:

$f \equiv g$  тогда и только тогда, когда  $f(x) \subseteq g(x)$  для всех  $x \in \mathcal{D}'$ .

В трактовке функций как абстрактных объектов нет ничего нового; но то, что они могут выступать в качестве объектов вычисления, требует проверки. Отношение  $\sqsubseteq$  на  $[\mathcal{D}' \rightarrow \mathcal{D}'']$  является первым шагом на пути доказательства этого, и он приводит нас к знакомому понятию конечной аппроксимации функций (прошу прощения, но у меня нет времени уточнять). Как только мы в этом убеждаемся, мы видим возможность *итерации* функциональных пространств; то же самое и в случае  $[[\mathcal{D}' \rightarrow \mathcal{D}'' ] \rightarrow \mathcal{D}'' ]$ . Это утверждение не так уж невероятно, как может показаться на первый взгляд, т. е. наша теория определяет  $f(x)$  как бинарную вычислимую функцию от переменной  $f$  и переменной  $x$ . И как операция  $f(x)$  может рассматриваться как элемент функционального пространства:

$$[[[\mathcal{D}' \rightarrow \mathcal{D}'' ] \times \mathcal{D}'] \rightarrow \mathcal{D}'' ] .$$

Это только начало теории таких операторов (или *комбинаторов*, как их называют Карри и Чёрч).

Усвоив все это, давайте попытаемся написать бесконечную итерацию функциональных пространств, взяв  $\mathcal{S}$  в качестве начального приближения. Пусть  $\mathcal{F}_0 = \mathcal{S}$  и  $\mathcal{F}_{n+1} = [\mathcal{F}_n \rightarrow \mathcal{S}]$ . Тогда  $\mathcal{F}_1 = [\mathcal{S} \rightarrow \mathcal{S}]$  и

$$\mathcal{F}_4 = [[[\mathcal{S} \rightarrow \mathcal{S}] \rightarrow \mathcal{S}] \rightarrow \mathcal{S}] .$$

Поверьте мне на слово, что все это очень конструктивно (именно *потому*, что мы использовали только непрерывные функции).

Совершенно очевидно, что существует естественная трактовка всей этой совокупности. Сначала  $\mathcal{S}$  «содержится» в прост-

пространстве  $[\mathcal{P} \rightarrow \mathcal{P}]$  как подпространство. Сопоставим каждому  $x \in \mathcal{P}$  соответствующую константную функцию из  $[\mathcal{P} \rightarrow \mathcal{P}]$ . Из нашего определения следует, что это соответствие сохраняет упорядоченность. Точно так же каждую  $f \in [\mathcal{P} \rightarrow \mathcal{P}]$  можно (грубо) аппроксимировать константой, скажем  $f(\perp)$  (это самый «лучший» элемент, предшествующий в отношении  $\sqsubseteq$  всем значениям  $f(x)$ ). Эту связь пространства и аппроксимации пространств будем обозначать  $\mathcal{P} \triangleleft [\mathcal{P} \rightarrow \mathcal{P}]$ .

Далее мы можем сказать, что

$$[\mathcal{P} \rightarrow \mathcal{P}] \triangleleft [[\mathcal{P} \rightarrow \mathcal{P}] \rightarrow \mathcal{P}],$$

но теперь по другой причине. Разобравшись с соотношением  $\mathcal{P} \triangleleft [\mathcal{P} \rightarrow \mathcal{P}]$ , рассмотрим теперь пространство более сложных структур вида  $\mathcal{F}_n$ . Для начала предположим  $f \in [\mathcal{P} \rightarrow \mathcal{P}]$ . Мы хотим отобразить  $f$  в следующее пространство, скажем  $i(f) \in [[\mathcal{P} \rightarrow \mathcal{P}] \rightarrow \mathcal{P}]$ . Для каждого элемента  $g \in [\mathcal{P} \rightarrow \mathcal{P}]$  по определению будем считать, что  $i(f)(g) \in \mathcal{P}$ . Теперь для  $g \in [\mathcal{P} \rightarrow \mathcal{P}]$  мы имеем обратную проекцию  $j(g) = g(\perp) \in \mathcal{P}$ . Если это наилучшая аппроксимация для  $g$ , которую мы можем получить в  $\mathcal{P}$ , мы обязаны определить

$$i(f)(g) = f(j(g)).$$

Тем самым мы получаем следующее отображение  $i: \mathcal{F}_1 \rightarrow \mathcal{F}_2$ . Соответствующую проекцию  $j: \mathcal{F}_2 \rightarrow \mathcal{F}_1$  определим аналогичным способом:

$$j(\phi)(x) = \phi(i(x)),$$

где  $\phi \in [[\mathcal{P} \rightarrow \mathcal{P}] \rightarrow \mathcal{P}]$ , а  $i(x) \in [\mathcal{P} \rightarrow \mathcal{P}]$  является константной функцией со значением, равным  $x$ . Используя те же рассуждения, определим  $i: \mathcal{F}_2 \rightarrow \mathcal{F}_3$  и  $j: \mathcal{F}_3 \rightarrow \mathcal{F}_4$  и так далее:

$$\mathcal{F}_0 \triangleleft \mathcal{F}_1 \triangleleft \mathcal{F}_2 \triangleleft \dots \triangleleft \mathcal{F}_n \triangleleft \mathcal{F}_{n+1} \triangleleft \dots$$

С учетом всего этого было бы жаль не перейти к пределу (на этот раз среди пространств), и это именно то, что я хотел бы вам сообщить. Что может следовать из утверждения, что существует пространство

$$\mathcal{F}_\infty = \lim_{n \rightarrow \infty} \mathcal{F}_n?$$

Так как  $\mathcal{F}_{n+1} = [\mathcal{F}_n \rightarrow \mathcal{P}]$  соответствует одному шагу итерации, то естественно предположить

$$\mathcal{F}_\infty = [\mathcal{F}_\infty \rightarrow \mathcal{P}]$$

(по крайней мере с точностью до изоморфизма). Это на самом деле так, но я могу только обрисовать основу (и обоснован-

ность) этого изоморфизма. Вначале отдельные пространства  $\mathcal{F}_n$  помещаются одно в другое, что не только создает башню пространств, но и трактует  $f(x)$  как алгебраическую операцию от *двух* переменных.  $\mathcal{F}_\infty$  является в точности результатом объединения  $\mathcal{F}_n$ ; т. е. *внутри* этих областей мы можем представить башни *функций*, каждая из которых аппроксимирует следующую (с использованием отображений  $i$  и  $j$ ), т. е. в  $\mathcal{F}_\infty$  эти башни имеют свои пределы. В случае усеченных башен мы можем утверждать, что каждое пространство  $\mathcal{F}_n \triangleleft \mathcal{F}_\infty$ .

Итак, почему же  $\mathcal{F}_\infty$  изоморфизм? Возьмем функцию (непрерывную) из  $[\mathcal{F}_\infty \rightarrow \mathcal{P}]$ . Так как эта функция непрерывна, то она определяется тем, как она воздействует на конечные уровни  $\mathcal{F}_n$ , т. е. она будет иметь все лучшую и лучшую аппроксимацию в  $[\mathcal{F}_n \rightarrow \mathcal{P}] = \mathcal{F}_{n+1}$ ; таким образом, аппроксимации «находятся» в конечных уровнях  $\mathcal{F}_\infty$ . Их предел должен дать нам снова ту же функцию из  $[\mathcal{F}_\infty \rightarrow \mathcal{P}]$ , с которой мы начали. Аналогично *любой* элемент из  $\mathcal{F}_\infty$  может рассматриваться как предел аппроксимирующих функций в пространствах  $[\mathcal{F}_n \rightarrow \mathcal{P}]$ . Возможно, необходимо проверить некоторые детали; однако при переходе к пределу нет действительного различия между  $\mathcal{F}_\infty$  и  $[\mathcal{F}_\infty \rightarrow \mathcal{P}]$ : бесконечный уровень функций высших порядков является его *собственным* функциональным пространством. (Как всегда, это является следствием непрерывности.)

$$\begin{aligned}\mathcal{F}_\infty \times \mathcal{F}_\infty &= [\mathcal{F}_\infty \rightarrow \mathcal{P}] \times [\mathcal{F}_\infty \rightarrow \mathcal{P}] \\ &= [\mathcal{F}_\infty \rightarrow \mathcal{P} + \mathcal{P}] \\ &= [\mathcal{F}_\infty \rightarrow \mathcal{P}] \\ &= \mathcal{F}_\infty\end{aligned}$$

Рис. 3. Первая цепочка изоморфизмов.

Под описанным здесь скрывается гораздо большая структура, чем я предполагал вначале. Рисунок 3 иллюстрирует цепочку изоморфизмов, показывающих, что  $\mathcal{F}$  получает многие из свойств  $\mathcal{P}$ , с которыми мы уже знакомы. Это верно по следующим соображениям. Во-первых, мы трактуем  $\mathcal{F}_\infty$  как функциональное пространство. Затем *парам функций* можно изоморфно поставить в соответствие функции, принимающие *пары значений*. Однако, как мы уже знаем,  $\mathcal{P} \times \mathcal{P} = \mathcal{P}$ . Последний переход как раз отображает функции на  $\mathcal{F}_\infty$  обратно в элементы из  $\mathcal{F}_\infty$ .

Используя изоморфизм на рис. 3, мы можем получить следующий результат, показанный на рис. 4. Рассуждения совершенно очевидны. Возьмем функцию, действующую из  $\mathcal{F}_\infty$  в  $\mathcal{F}_\infty$ . Значения этой функции можно интерпретировать как функции.

$$\begin{aligned}
 \mathcal{F}_\infty \rightarrow \mathcal{F}_\infty &= [\mathcal{F}_\infty \rightarrow [\mathcal{F}_\infty \rightarrow \mathcal{P}]] \\
 &= [[\mathcal{F}_\infty \times \mathcal{F}_\infty] \rightarrow \mathcal{P}] \\
 &= [\mathcal{F}_\infty \rightarrow \mathcal{P}] \\
 &= \mathcal{F}_\infty.
 \end{aligned}$$

Рис. 4. Вторая цепочка  
изоморфизмов.

Но учтем, что функция, значениями которой являются функции, есть в точности *функция от двух аргументов* (с точностью до изоморфизма). Как мы уже видели на рис. 3,  $\mathcal{F}_\infty \times \mathcal{F}_\infty = \mathcal{F}_\infty$ , поэтому мы получаем последний переход (с точностью до изоморфизма).

Мы обрисовали, почему  $\mathcal{F}_\infty$ , пространство функций бесконечного порядка, является моделью  $\lambda$ -исчисления.  $\lambda$ -исчисление — язык (здесь не рассматривавшийся), в котором каждый терм может *одновременно* обозначать как аргумент (значение), так и функцию. *Формальные* детали довольно просты, а *семантические* — как раз те, которые мы рассматривали: каждый элемент пространства  $\mathcal{F}_\infty$  может в то же время быть элементом пространства  $[\mathcal{F}_\infty \rightarrow \mathcal{F}_\infty]$ ; таким образом,  $\mathcal{F}_\infty$  дает модель, которая является одной из многих.

Не претендуя на точность, мы в общих чертах обрисовали денотационную семантику для чисто процедурного языка (а также для пар и т. д., см. рис. 2). В перечисленных ниже работах по реальным языкам программирования указаны все остальные черты (присваивание, упорядочение, объявления и т. д.). В них установлено, что метод семантического определения действительно работает. Я надеюсь, что и вы заинтересуетесь им.

## ЛИТЕРАТУРА

1. Bohm C., Ed.  $\lambda$ -Calculus and Computer Science Theory. Lecture Notes in Computer Science, Vol. 37. Springer-Verlag, New York, 1975.
2. Clark K. L., Cowell D. F. Programs, Machines and Computation. McGraw-Hill, New York, 1976.
3. Crossley J. N., Ed. Algebra and Logic. Papers from the 1974 Summer Res. Inst. Australian Math. Soc., Monash U. Clayton, Victoria, Australia.
4. Donahue J. E. Complementary Definitions of Programming Language Semantics. Lecture Notes in Computer Science, Vol. 42, Springer-Verlag, 1976.
5. Eilenberg S. Automata, Languages, and Machines. Academic Press, New York, 1974.
6. van Emden M. H., Kowalski R. A. The semantics of predicate logic as a programming language. J. ASM 23, 4 (Oct. 1976), 733—742.
7. Manes E. G., Ed. Category Theory Applied to Computation and Control. First Int. Symp. Lecture Notes in Computer Science, Vol. 25 Springer-Verlag, New York, 1976.

8. Manna Z. Mathematical Theory of Computation. McGraw-Hill, New York, 1974.
9. Milne R., Strachey C. A. Theory of Programming Language Semantics. Chapman and Hall, London, and Wiley, New York, 2 Vols., 1976.
10. Plotkin G. D. A powerdomain construction. SIAM J. Computng. 5 (1976), 452—487.
11. Rabin M. O., Scott D. S. Finite automata and their decision problems. IBM J. Res. and Develop. 3 (1959), 114—125.
12. Scott D. S. Data types as lattices. SIAM J. Computng. 5 (1976), 522—587.
13. Stoy J. E. Denotational Semantics — The Scott — Strachey Approach to Programming Language Theory. M. I. T. Press, Cambridge, Mass.
14. Tennent R. D. The denotational semantics of programming languages. Comm. ACM 19, 8 (Aug. 1976), 437—453.



1977

# **Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ**

*Дж. Бэкус*

Исследовательская лаборатория IBM,  
Сан-Хосе

Тьюринговская премия АСМ за 1977 г. была присуждена Дж. Бэкусу на ежегодной конференции АСМ в Сиэтле 17 октября. Представляя лауреата, председатель Комитета по премиям Дж. Е. Семмет сделал следующие замечания и зачитал часть заключительного представления. (Полный текст представления опубликован в выпуске Communications of АСМ за сентябрь 1977 г., с. 681.)

«Вероятно, в этом зале не найдется человека, который не слышал бы о Фортране, и большинство из вас, наверное, использовали его хотя бы однажды или по крайней мере смотрели из-за плеча кого-то, кто писал программу на Фортране. Наверное, почти столько же людей слышали комбинацию букв БНФ, хотя не обязательно знают, что означают эти буквы. Так вот, «Б» означает Бэкуса, а остальные буквы объясняются в формальном представлении комитета. По моему мнению, эти два достижения входят в число полдюжины наиболее важных вкладов в информатику, и оба принадлежат Джону Бэкусу (который в случае Фортрана привлек к работе нескольких коллег). Именно за эти достижения он получает Тьюринговскую премию этого года».

Вкратце, он представлен к премии за «глубокий, оказавший решающее влияние и долговременный вклад в проектирование практических систем программирования высокого уровня, особенно проявившийся в его работе по Фортрану и в плодотворной публикации формальных процедур для спецификаций языков программирования».

Основная часть полного представления к премии сформулирована так:

«...Бэкус возглавлял небольшую группу IBM в Нью-Йорке в начале 50-х гг. Первым результатом работы этой группы явился язык высокого уровня для научных и технических вычислений, названный Фортраном. Та же группа разработала

первую систему для трансляции программ на Фортране в машинный код. Они применяли новые методы оптимизации для генерации быстрых программ в машинном коде. Для этого языка разрабатывалось много других компиляторов, сначала на машинах IBM, а впоследствии практически на любых компьютерах. В 1966 г. Фортран был принят как национальный стандарт США.

В конце 50-х гг. Бэкус работал в международном комитете, который разработал Алгол-58 и следующую версию Алгол-60. Язык Алгол и производные от него компиляторы получили широкое признание в Европе как средство разработки программ и как формальное средство публикации алгоритмов, на которых основываются программы.

В 1959 г. Бэкус представил на конференцию Юнеско в Париже доклад о синтаксисе и семантике предлагаемого международного алгебраического языка. В этом докладе он впервые применил формальный метод спецификации синтаксиса языков программирования. Эта формальная нотация получила известность как БНФ, что означает «Бэкуса нормальная форма» или «Бэкуса — Наура форма» в знак признания дальнейшего вклада П. Наура из Дании.

Итак, Бэкус внес значительный вклад как в прагматику решения задач на компьютерах, так и в теорию взаимосвязи между искусственными языками и вычислительной лингвистикой. Фортран остается одним из наиболее широко используемых во всем мире языков программирования. Теперь почти все языки программирования описываются с помощью некоторого вида формального определения синтаксиса».

Обычные языки программирования постоянно растут по объему, но не по своей выразительной силе. Неустранимые дефекты на самом фундаментальном уровне делают их тучными и слабыми: их примитивный стиль программирования «слово за словом» восходит к их общему прародителю — компьютеру фон Неймана; в них семантика тесно сплетается с переходами состояний; они разделяют программирование на мир выражений и мир операторов; они не дают возможности эффективно использовать мощные приемы комбинирования для построения новых программ из уже существующих, и в них недостает полезных математических свойств для рассуждений о программах.

Другой, функциональный стиль программирования находит применение в комбинационных формах создания программ. Функциональные программы оперируют со структурированными данными, часто оказываются неповторяемыми и нерекурсивными, конструируются иерархически, не именуют свои аргументы и не требуют сложной техники описаний процедур в целях универсальности. В комбинационных формах программы высокого уровня могут использоваться для построения программ еще более высокого уровня в таком стиле, который недоступен для традиционных языков.

С функциональным стилем программирования ассоциируется алгебра программ, в которой переменными служат программы, а операциями являются комбинационные формы. Эта алгебра может служить для преобразования

программ и для решения уравнений, в которых «неизвестными» являются программы, причем процесс решения во многом аналогичен преобразованиям уравнений в курсе высшей алгебры. Применяемые преобразования задаются алгебраическими законами и выполняются на том же языке, на котором пишутся программы. Комбинационные формы выбираются не только из-за их программистской выразительности, но и из-за мощи соответствующих им алгебраических законов. Общие теоремы алгебры задают детали поведения и условия завершения для больших классов программ.

В новом классе вычислительных систем стиль функционального программирования используется и в языке программирования, и в правилах перехода между состояниями. В отличие от языков фон Неймана в этих системах семантика слабо связана с состояниями — в течение большого вычисления происходит только одна смена состояний.

## ВВЕДЕНИЕ

Я глубоко признателен за оказанную мне честь приглашением от АСМ прочесть тьюринговскую лекцию 1977 г. и опубликовать этот отчет о ней с подробностями, обещанными в лекции. Читателям, желающим ознакомиться с резюме этой публикации, следует обратиться к последнему разделу (разд. 16).

### 1. ТРАДИЦИОННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ: ТУЧНОСТЬ И ВЯЛОСТЬ

По-видимому, с языками программирования происходит что-то неладное. Всякий новый язык включает с небольшими изменениями все свойства своих предшественников плюс кое-что еще. Руководства по некоторым языкам занимают более 500 страниц; в других случаях сложные описания втиснуты в менее пространственные руководства с помощью неудобопонятного формализма. Министерство обороны планирует в настоящее время стандарт спроектированного комитетом языка, для которого может потребоваться руководство примерно в 1000 страниц. В каждом новом языке заявляются новые и удобные возможности, например сильная типизация или структурированные управляющие операторы, но фактически дела обстоят так, что лишь немногие языки снижают затраты на программирование или повышают его надежность в достаточной степени, чтобы оправдать стоимость создания этих языков и обучения их использованию.

Поскольку большие увеличения объема приводят лишь к малому росту мощности, сохраняется популярность меньших, более изящных языков, например Паскаля. Однако имеется настоящая необходимость в эффективной методологии, которая могла бы нам размышлять о программах, и ни один из тради-

ционных языков даже не подступился к удовлетворению этой потребности. В действительности традиционные языки создают ненужные помехи для наших суждений о программах.

В течение двадцати лет языки программирования неизменно развивались в одном и том же направлении, пока не дошли до нынешнего состояния «ожирения»; в результате изучение и изобретение языков программирования в значительной степени потеряли свою привлекательность. Напротив, теперь это излюбленная область деятельности для тех, кто предпочитает возиться с пухлыми перечнями подробностей вместо того, чтобы бороться за новые идеи. Дискуссии о языках программирования часто напоминают средневековые диспуты о числе ангелов, которые могут разместиться на кончике иглы, а не волнующие споры о фундаментально различных понятиях.

Многие творчески одаренные исследователи переключились с изобретения языков на изобретение средств их описания. К сожалению, им приходилось применять свои изящные новые средства преимущественно к изучению бородавок и родинок существующих языков. Достойно удивления, почему столь многие из нас, изучив отвратительные структуры типов традиционных языков с помощью изящного инструментария, разработанного Д. Скоттом, пассивно сохраняют верность этим структурам вместо того, чтобы энергично искать новые структуры.

Данная статья преследует две цели: во-первых, показать, что фундаментальные недостатки традиционных языков делают их выразительную слабость и их злокачественное разрастание неизбежными, и, во-вторых, предложить альтернативные пути исследований в направлении проектирования новых видов языков.

## 2. МОДЕЛИ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Основой всякого языка программирования является модель вычислительной системы, которой управляют программы на этом языке. Некоторые модели являются чистейшими абстракциями, некоторые представляются аппаратным обеспечением, а другие — компилирующими или интерпретирующими программами. Прежде чем вплотную заняться изучением традиционных языков, полезно дать краткий обзор существующих моделей как введение в текущее разнообразие альтернатив. Грубая классификация существующих моделей может основываться на намеченных ниже критериях.

## 2.1. КРИТЕРИИ ОЦЕНКИ МОДЕЛЕЙ

**2.1.1. Основания.** Существует ли изящное и точное математическое описание модели? Годится ли оно для доказательства полезных фактов относительно поведения модели? Или же модель настолько сложна, что ее описание является громоздким, и математик мало что может с ним сделать?

**2.1.2. Историческая чувствительность.** Включает ли модель понятие памяти, благодаря которому одна программа может запасать информацию, которая может повлиять на поведение более поздней программы? Иначе говоря, является ли модель исторически чувствительной?

**2.1.3. Тип семантики.** Осуществляет ли программа последовательные преобразования состояний (не являющихся программами) до тех пор, пока не будет достигнуто завершающее состояние (семантика смены состояний)? Являются ли состояния простыми или сложными? Может ли «программа» последовательно сводиться к более простым «программам», чтобы в конечном итоге превратиться в «программу в нормальной форме», которая представляет собой результат последовательного сведения (редукционная семантика)?

**2.1.4. Ясность и концептуальная полезность программ.** Являются ли программы модели ясными выражениями процесса или вычисления? Воплощаются ли в них концепции, помогающие нам формулировать утверждения и рассуждения о процессах?

## 2.2. КЛАССИФИКАЦИЯ МОДЕЛЕЙ

С помощью перечисленных критериев мы можем грубо охарактеризовать три класса моделей для вычислительных систем: простые операционные модели, аппликативные модели и модели фон Неймана.

**2.2.1. Простые операционные модели.** Примеры: машины Тьюринга, различные автоматы. *Основания:* точные и полезные. *Историческая чувствительность:* обладают памятью и исторической чувствительностью. *Семантика:* смена состояний с очень простыми состояниями. *Ясность программ:* программы неясные и концептуально бесполезные.

**2.2.2. Аппликативные модели.** Примеры: лямбда-исчисление Чёрча [5], система комбинаторов Карри [6], чистый Лисп [17], системы функционального программирования, описываемые в этой статье. *Основания:* точные и полезные. *Историческая чувствительность:* нет памяти, нет исторической чувствительности. *Семантика:* редукционная семантика, нет состояний. *Ясность*

*программ*: программы могут быть ясными и функционально полезными.

**2.2.3. Модели фон Неймана.** Примеры: компьютеры фон Неймана, традиционные языки программирования. *Основания*: сложные, громоздкие, бесполезные. *Историческая чувствительность*: обладают памятью, исторически чувствительны. *Семантика*: смена состояний со сложными состояниями. *Ясность программ*: программы могут быть умеренно ясными и не очень полезными концептуально.

Приведенная выше классификация, возможно, является грубой и спорной. Некоторые недавно появившиеся модели не удастся без труда подогнать к какой-либо из этих категорий. Например, языки потоков данных, разработанные Эрвидном и Гостлоу [1], Деннисом [7], Косински [13] и другими исследователями, отчасти обладают свойствами, позволяющими отнести их к классу простых операционных моделей, но их программы яснее, чем в прежних моделях из этого класса, и, по-видимому, можно утверждать, что некоторые из них обладают редуцированными семантиками. Во всяком случае, данная классификация будет служить грубой картой изучаемой территории. Мы сосредоточим внимание на аппликативных моделях и моделях фон Неймана.

### 3. КОМПЬЮТЕРЫ ФОН НЕЙМАНА

Для того чтобы понимать проблематику традиционных языков программирования, нам нужно сначала исследовать их интеллектуального прародителя, компьютер фон Неймана. Что такое компьютер фон Неймана? Когда фон Нейман и другие задумывали его более тридцати лет назад, это была изящная, практичная и объединяющая идея, которая упрощала ряд существовавших тогда инженерных и программистских задач. Хотя условия, породившие архитектуру этого компьютера, с тех пор радикально изменились, тем не менее мы по-прежнему идентифицируем понятие «компьютера» с этой концепцией тридцатилетней давности.

В своей простейшей форме компьютер фон Неймана состоит из трех частей: центрального процессорного устройства (ЦПУ), памяти и соединительной шины, которая может за один шаг передавать только одно слово между ЦПУ и памятью (и посылать некий адрес в память). Я предлагаю называть эту шину «*бутылочным горлышком*» (узким местом) *фон Неймана*. Задача программы состоит в том, чтобы неким существенным образом изменить содержимое памяти; если считать, что эта задача должна быть выполнена исключительно перекачивани-

ем одиночных слов туда и обратно через узость фон Неймана, то становится ясной причина такого названия.

Ирония ситуации состоит в том, что большую часть потока через эту узость составляют не полезные данные, а всего лишь имена данных, а также операции и данные, служащие лишь для вычисления таких имен. Прежде чем слово можно будет послать через шину, его адрес должен находиться в ЦПУ; поэтому он должен либо быть послан через шину из памяти, либо генерироваться посредством некоторой операции ЦПУ. Если адрес посылается из памяти, то *адрес этого адреса* должен либо быть послан из памяти, либо генерироваться в ЦПУ, и т. д. С другой стороны, если адрес генерируется в ЦПУ, он должен генерироваться либо по фиксированному правилу (например, «добавить 1 к счетчику программы»), либо по команде, которая была послана через шину, в последнем случае ее адрес нужно было прежде послать... и т. д.

Разумеется, должен существовать менее примитивный способ внесения в память больших изменений, чем мельтешение множества слов туда и обратно через узость фон Неймана. Эта шина является не только узким местом для потока данных задачи, но, что более важно, и интеллектуальной узостью, которая привязывает нас к мышлению «слово за словом» вместо того, чтобы вдохновлять нас на мышление более крупными концептуальными частями решаемой задачи. Итак, программирование в основном сводится к планированию и спецификации огромного потока слов через узость фон Неймана, причем большая часть этого потока состоит не из самих значащих данных, а из сведений о том, где их искать.

#### 4. ЯЗЫКИ ФОН НЕЙМАНА

Обычные языки программирования в основном являются высокоуровневыми, сложными версиями компьютера фон Неймана. Наша вера тридцатилетней давности, что существует только один вид компьютера, стала основой нашей уверенности в том, что существует только один вид языка программирования — традиционный язык фон Неймана. При всей существенности различия между Фортраном и Алголом-58 оно имеет меньше значения, чем тот факт, что оба этих языка основываются на программистском стиле компьютера фон Неймана. Хотя я упоминаю традиционные языки как «языки фон Неймана», чтобы отметить их первоисточник и стиль, я, конечно, не упрекаю великого математика за их сложность. Действительно, многие могли бы сказать, что я сам до некоторой степени несу ответственность за эту проблему.

В языках программирования фон Неймана переменные используются для имитации ячеек памяти компьютера; операторы управления выражают его команды передачи управления и проверки, а операторы присваивания имитируют вызов содержимого ячейки, запоминание и арифметику. Оператор присваивания представляет собой узость фон Неймана для языков программирования и вынуждает нас думать в терминах «слово за словом» примерно таким же образом, как на наше мышление воздействует существование в компьютере шины обмена данными между памятью и ЦПУ.

Рассмотрим типичную программу. Ее основу составляет ряд операторов присваивания, содержащих некоторые переменные с индексами. Каждый оператор присваивания порождает результат, состоящий из одного слова. Программа должна организовывать многократное выполнение этих операторов со сменой значений индексов, чтобы произвести желаемое итоговое изменение в памяти, поскольку она связана необходимостью изменять каждый раз только одно слово. Таким образом, программист имеет дело с потоком слов через узость присваиваний в соответствии с тем, как он проектирует вложенность управляющих операторов для обеспечения необходимых повторений.

Кроме того, оператор присваивания расщепляет программирование на два мира. Первый мир включает в себя правые части операторов присваивания. Это упорядоченный мир выражений, мир с полезными алгебраическими свойствами (если не учитывать того, что эти свойства часто нарушаются побочными эффектами). Это тот мир, в котором происходит большинство полезных вычислений.

Второй мир традиционных языков программирования — это мир операторов. Первичным оператором в этом мире является сам оператор присваивания. Все остальные операторы языка существуют для того, чтобы создать возможность выполнения вычисления, которое должно основываться на этой примитивной конструкции: на операторе присваивания.

Этот мир операторов неупорядочен, и у него мало полезных математических свойств. Структурное программирование можно считать скромной попыткой внести некий порядок в этот хаотический мир, но оно в малой степени способствует разрешению тех фундаментальных проблем, которые вносятся «словным» стилем программирования фон Неймана с его примитивным использованием циклов, индексов и разветвлений потока управления.

Наша фиксация на языках фон Неймана сохранила преобладание компьютера фон Неймана, а наша зависимость от него сделала не-фон-неймановские языки неэкономичными и ограничила их развитие. Отсутствие законченных, эффективных стилей



программирования, опирающихся на принципы, отличные от принципов фон Неймана, обезоружило проектировщиков интеллектуальных основ новых компьютерных архитектур. (Эта проблема кратко обсуждается в разд. 15.)

Аппликативные вычислительные системы не стали основой проектирования компьютеров главным образом из-за отсутствия в них памяти и исторической чувствительности. Кроме того, в большинстве аппликативных систем в качестве основной операции используется операция подстановки лямбда-исчисления. Эта операция обладает фактически неограниченной выразительной мощностью, но ее полная и эффективная реализация весьма затруднительна для проектировщиков компьютеров. К тому же при попытках оснастить аппликативные системы памятью и повысить их эффективность на компьютерах фон Неймана возникла тенденция к погружению этих систем в большие системы фон Неймана. Например, чистый Лисп часто становится подмножеством больших расширений, обладающих многими свойствами систем фон Неймана. Получающиеся в результате сложные системы мало что могут подсказать разработчику компьютеров.

## 5. СРАВНЕНИЕ ПРОГРАММ ФОН НЕЙМАНА С ФУНКЦИОНАЛЬНЫМИ ПРОГРАММАМИ

Чтобы получить более детальное представление о некоторых недостатках языков фон Неймана, сравним обычную программу вычисления внутреннего произведения с соответствующей функциональной программой, написанной на простом языке, который будет детализирован впоследствии.

### 5.1. ПРОГРАММА ФОН НЕЙМАНА ДЛЯ ВНУТРЕННЕГО ПРОИЗВЕДЕНИЯ

```
c: = 0
for i: 1 step 1 until n do
c: = c + a[i] × b[i]
```

Заслуживают упоминания следующие свойства этой программы.

(а) Ее операторы действуют на скрытые «состояния» в соответствии со сложными правилами.

(б) Она не является иерархической. За исключением правой части оператора присваивания, она не конструирует сложных объектов из более простых. (Впрочем, большие программы часто делают это.)

(в) Она динамическая и итеративная. Человек должен мысленно исполнить ее, чтобы понять ее работу.

(г) Она проводит вычисления пословно, повторяя (присваивание) и модифицируя (переменную  $i$ ).

(д) Часть данных (число  $n$ ) содержится в программе; поэтому ей недостает общности и она работает только с векторами длины  $n$ .

(е) Она именует свои аргументы; ее можно использовать только для векторов  $a$  и  $b$ . Чтобы она стала общей, требуется описание процедуры. При этом возникают сложные проблемы (например, вызов по имени вместо вызова по значению).

(ж) Ее операции «внутреннего хозяйства» представляются символами, помещенными в разных местах (в операторе **for** и в индексах оператора присваивания). Из-за этого невозможно сосредоточить операции внутреннего хозяйства, которые являются наиболее общими из всех операций, в единых, мощных, широко используемых операциях. Таким образом, при программировании таких операций всегда приходится начинать снова «в лоб», выписывая «**for**  $i := \dots$ » и «**for**  $j := \dots$ », а затем операторы присваивания, пестрящие индексами  $i$  и  $j$ .

## 5.2. ФУНКЦИОНАЛЬНАЯ ПРОГРАММА ДЛЯ ВНУТРЕННЕГО ПРОИЗВЕДЕНИЯ

**Def** Внутреннее произведение = (Вставить  $+$ )  $\circ$  (Применить ко всем  $\times$ )  $\circ$  Транспозиция или в сокращенной форме

**Def**  $IP = (/+) \circ (\alpha \times) \circ \text{Trans}$ .

Композиция ( $\circ$ ), Вставить ( $/$ ) и Применить ко всем ( $\alpha$ ) являются *функциональными формами*, которые комбинируют существующие функции для образования новых. Так,  $f \circ g$  — это функция, получаемая применением сначала  $g$ , затем  $f$ , а  $\alpha f$  — функция, получаемая применением  $f$  к каждому *слагаемому* аргумента. Если мы пишем  $f:x$  для обозначения результата применения  $f$  к объекту  $x$ , то можем объяснить каждый шаг вычисления внутреннего произведения  $IP$  применительно к паре векторов  $\langle 1, 2, 3 \rangle$ ,  $\langle 6, 5, 4 \rangle$  следующим образом:

$IP: \langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle =$

Описание  $IP \Rightarrow (/+) \circ (\alpha \times) \circ \text{Trans}: \langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle$

Результат композиции,  $\Rightarrow (/+): ((\alpha \times): \text{Trans}: \langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle)$

Применение транспозиции  $\Rightarrow (/+): ((\alpha \times): \langle 1, 6 \rangle, \langle 2, 5 \rangle, \langle 3, 4 \rangle)$

Результат Применить ко всем,  $\alpha \Rightarrow (/+): \langle \times: \langle 1, 6 \rangle, \times: \langle 2, 5 \rangle, \times: \langle 3, 4 \rangle \rangle$

Применение  $\times \Rightarrow (/+): \langle 6, 10, 12 \rangle$

Результат Вставить, /  $\Rightarrow + : \langle 6, + : \langle 10, 12 \rangle$

Применение  $+$   $\Rightarrow + : \langle 6, 22 \rangle$

Применение  $+$  снова  $\Rightarrow 28$

Сравним свойства этой программы со свойствами программы фон Неймана.

(а) Она оперирует только своими аргументами. Здесь нет скрытых состояний или сложных правил перехода. Имеются только два вида правил, одно для применения функции к ее аргументу, а другое для получения функции, обозначаемой такой функциональной формой, как композиция  $f \circ g$  или Применить ко всем,  $\alpha f$ , если известны параметры форм, т. е. функции  $f$  и  $g$ .

(б) Она является иерархической, поскольку строится из трех более простых функций ( $+$ ,  $\times$ , Trans) и трех функциональных форм  $f \circ g$ ,  $\alpha f$  и  $f/g$ .

(в) Она является статической и неитеративной в том смысле, что ее структура удобна для ее понимания без мысленного исполнения. Например, если некто понимает действие форм  $f \circ g$  и  $\alpha f$  и функций  $\times$  и Trans, то он понимает действие  $\alpha \times$  и  $(\alpha \times) \circ \text{Trans}$  и т. д.

(г) Она оперирует целыми концептуальными блоками, а не словами; она состоит из трех этапов, ни один этап не повторяется.

(д) Она не включает данных; она является вполне общей; она работает для любой пары векторов одинаковой размерности.

(е) Она не именует свои аргументы; она применима к любой паре векторов без всяких описаний процедур или сложных правил подстановок.

(ж) Она использует формы и функции внутреннего хозяйства, которые универсально полезны во многих других программах; на самом деле только операции  $+$  и  $\times$  не имеют отношения к внутреннему хозяйству. Эти формы и функции могут комбинироваться с другими, создавая операции внутреннего хозяйства более высокого уровня.

Раздел 14 содержит набросок некоей системы, спроектированной для того, чтобы сделать такой функциональный стиль программирования доступным в простой по структуре, исторически чувствительной системе, но потребуется еще много труда, прежде чем этот функциональный стиль сможет стать основой изящных и практичных языков программирования. В настоящее время проведенные выше сравнения выявляют ряд серьезных недостатков языков фон Неймана и могут служить отправной точкой для усилий по преодолению их нынешней точности и вялости.

## 6. СТРУКТУРЫ ЯЗЫКОВ В СРАВНЕНИИ С ИЗМЕНЯЕМЫМИ ЧАСТЯМИ

Будем различать две части языка программирования. Первой является *структура*, которая задает общие правила системы, а вторая — это *изменяемая часть*, существование которой подразумевается структурой, но конкретное поведение которой не специфицируется. Например, оператор `for` и почти все остальные операторы являются частью структуры Алгола, а библиотечные функции и описываемые пользователем процедуры представляют собой изменяемую часть. Итак, структура языка описывает его фиксированные свойства и общую среду для его изменяемых свойств.

Теперь предположим, что язык обладает небольшой структурой, которая может объединять много разнообразных мощных средств целиком в качестве изменяемых частей. Тогда такая структура могла бы поддерживать много разных выразительных средств и стилей, не изменяясь при этом сама. В отличие от этой приятной возможности языки фон Неймана, по-видимому, всегда обладают огромной структурой и весьма ограниченными изменяемыми частями. Почему так получается? Ответ на этот вопрос затрагивает проблемы, связанные с языками фон Неймана.

Первая проблема возникает из-за стиля фон Неймана пословного программирования, который требует, чтобы слова сновали взад и вперед к состоянию и от него, совсем как в потоке через узость фон Неймана. Таким образом, язык фон Неймана должен обладать семантикой, тесно увязанной с состоянием, причем всякая подробность вычислений изменяет состояние. Вследствие такой тесной связи семантики с состояниями всякая деталь каждого свойства должна быть встроена в состояние и в его правила перехода.

Итак, каждое свойство языка фон Неймана должно быть закодировано с ошеломляющими подробностями в структуре языка. Более того, многие сложные свойства нужно подпирать слабым фундаментом пословного стиля. Результатом является неизбежно жесткая и огромная структура языка фон Неймана.

## 7. ИЗМЕНЯЕМЫЕ ЧАСТИ И КОМБИНАЦИОННЫЕ ФОРМЫ

Вторая проблема, связанная с языками фон Неймана, состоит в слишком малой выразительной силе их изменяемых частей. Красноречивым доказательством этого являются их раблезианские размеры. К тому же, если бы проектировщик знал, что все усложненные свойства, которые он теперь встраивает в структуру, могли быть впоследствии добавлены в качестве

изменяемой части, он не стал бы столь усердно втискивать их в структуру.

Возможно, самым важным аспектом обеспечения мощи изменяемой части языка является доступность комбинационных форм, которые в общем случае могут служить для построения новых процедур из старых. Языки фон Неймана обеспечивают только примитивные комбинационные формы, а структура фон Неймана создает препятствия для их полноценного использования.

Одним из препятствий применению комбинационных форм является водораздел между миром выражений и миром операторов в языках фон Неймана. Естественно, функциональные формы принадлежат к миру выражений, но вне зависимости от их мощи они могут строить только такие выражения, которые порождают однословный результат. И именно в мире операторов эти однословные результаты должны сочетаться в итоговом результате. Сочетание отдельных слов — это не то, о чем нам на самом деле следовало бы думать, тем не менее это большая часть программирования любой задачи на языках фон Неймана. Чтобы помочь сборке общего результата из отдельных слов, эти языки обеспечивают некоторые примитивные комбинационные формы в мире операторов — операторы **for**, **while** и **if — then — else**, но водораздел между двумя мирами мешает тому, чтобы комбинационные формы в каждом из миров обрели полную силу, которую они могли бы получить в неразделенном мире.

Второе препятствие применению комбинационных форм в языках фон Неймана состоит в использовании в них хитроумных соглашений именования, которые дополнительно усложняются правилами подстановок, требуемых в процедурах вызова. Для каждой из них нужно, чтобы в структуру языка был встроен сложный механизм, обеспечивающий возможность правильной интерпретации простых переменных, переменных с индексами, указателей, имен файлов, имен процедур, формальных параметров, вызываемых по значению, формальных параметров, вызываемых по имени, и т. д. Все эти имена, соглашения и правила затрудняют использование простых комбинационных форм.

## 8. ЯЗЫК APL В СРАВНЕНИИ С ПОСЛОВНЫМ ПРОГРАММИРОВАНИЕМ

Сказав так много о пословном программировании, я должен теперь что-нибудь сказать о языке APL [12]. Мы очень обязаны К. Айверсону, который показал нам, что существуют программы, не являющиеся пословными и не зависящие от лямб-

да-выражений, и познакомил нас с применением новых функциональных форм. А поскольку операторы присваивания языка APL могут запоминать массивы, то воздействие их функциональных форм выходит за рамки одиночного присваивания.

Однако, к сожалению, язык APL все же расщепляет программирование на мир выражений и мир операторов. Поэтому старание писать однострочные программы частично обусловлено желанием оставаться в более упорядоченном мире выражений. В языке APL имеются ровно три функциональные формы, называемые внутренним произведением, внешним произведением и сведением. Иногда их трудно использовать, их количество недостаточно, и их употребление ограничивается миром выражений.

Наконец, семантика языка APL по-прежнему тесно связана с состоянием. Поэтому, несмотря на большую простоту и мощь языка, его структура характеризуется сложностью и жесткостью языков фон Неймана.

## 9. НЕДОСТАТОЧНОСТЬ ПОЛЕЗНЫХ МАТЕМАТИЧЕСКИХ СВОЙСТВ В ЯЗЫКАХ ФОН НЕЙМАНА

До сих пор мы обсуждали большой размер и негибкость языков фон Неймана; другим их существенным неудобством является недостаточность математических свойств и вызываемая ими затруднительность рассуждений о программах. Несмотря на изобилие отменных публикаций с доказательствами фактов о программах, языки фон Неймана почти полностью лишены полезных в этом отношении свойств и обладают множеством свойств, являющихся помехами (например, побочные эффекты, различные имена для одних и тех же объектов).

Денотационная семантика [23] и ее обоснование [20, 21] обеспечивают исключительно полезное понимание скрытых в программах областей определения и функциональных пространств. При применении к «практичному» языку (например, к языку «рекурсивных программ» из [16]) ее основания дают эффективный инструментарий для описания языка и доказательства свойств программ. С другой стороны, при применении к языку фон Неймана она предоставляет точное семантическое описание и полезна для обнаружения слабых мест в языке. Однако сложность языка отражается в сложности описания, которое представляет собой ошеломляющее нагромождение правил подстановок, областей определения, функций и уравнений, лишь немногим более удобных для доказательства фактов о программах, чем стандартное руководство по языку, поскольку в таком руководстве все-таки меньше двусмысленностей.

Аксиоматическая семантика [11] в точности воспроизводит изящные свойства программ фон Неймана (т. е. преобразования на состояниях) в виде преобразований на предикатах. Тем самым изменяется не сама пословная итеративная игра, а только поле для игры. Сложность этой аксиоматической игры в доказательстве фактов о программах фон Неймана делает успехи ее участников еще более восхитительными. Кроме изобретательности, их успех основывается на двух дополнительных факторах. Во-первых, игра ограничивается малыми, слабыми подмножествами языков фон Неймана со значительно более простыми состояниями, чем в реальных языках. Во-вторых, новое поле для игры (предикаты и их преобразования) более богато, упорядоченно и эффективно, чем прежнее (состояния и их преобразования). Но ограничив богатство этой игры и перенеся ее на более эффективную область, мы утрачиваем возможность работать с реальными программами (с неизбежными сложностями вызовов процедур и несовпадений имен) и к тому же не исключим неуклюжие свойства основного стиля фон Неймана. По мере того как аксиоматическая семантика обобщается, чтобы покрывать более обширные подмножества типичного языка фон Неймана, она начинает терять свою эффективность из-за возрастания требуемой сложности.

Итак, денотационная и аксиоматическая семантики представляют собой описательные формализмы, опирающиеся на изящные и мощные понятия, но использование их для описания языка фон Неймана не может породить изящного и мощного языка, подобного тому, как употребление изящных и мощных машин для построения игрушечных автомобильчиков не может породить изящный и мощный современный автомобиль.

В любом случае для доказательства фактов о программах используется язык логики, а не язык программирования. Доказательства ведут речь о программах, но не могут использовать их непосредственно, потому что аксиомы языков фон Неймана являются столь неконструктивными. Напротив, многие обычные доказательства выводятся алгебраическими методами. Для этих методов требуется язык, обладающий определенными алгебраическими свойствами. Потом алгебраические законы могут применяться довольно механически для преобразования проблемы в ее решение. Например, для решения уравнения

$$ax + bx = a + b$$

относительно  $x$  (при условии, что  $a + b \neq 0$ ) мы механически применяем последовательно законы дистрибутивности, тождества и сокращения, чтобы получить

$$(a + b)x = a + b$$

Можно ли освободить программирование от стиля фон Неймана?

$$(a+b)x = (a+b)1$$

$$x = 1.$$

Итак, мы доказали, что  $x=1$ , не отходя от «языка» алгебры. Языки фон Неймана со своим нелепым синтаксисом представляют мало таких возможностей для преобразования программ.

Как мы увидим, программы могут быть выражены на языке, с которым связана некоторая алгебра. Эта алгебра может служить для преобразования программ и для решения некоторых уравнений, в которых «неизвестными» являются программы, примерно таким же образом, как решаются уравнения в высшей алгебре. В алгебраических преобразованиях и доказательствах используется язык самих программ, а не язык логики для изложения фактов о программах.

## 10. КАКОВЫ АЛЬТЕРНАТИВЫ ДЛЯ ЯЗЫКОВ ФОН НЕЙМАНА?

Прежде чем переходить к рассмотрению альтернатив языков фон Неймана, замечу, что меня огорчает необходимость проведенного выше негативного и не слишком точного обсуждения этих языков. Но благодушное принятие большинством из нас этих громоздких, слабых языков слишком долго озадачивало и расстраивало меня. Я расстраиваюсь из-за того, что это принятие поглотило огромные усилия, направленные на то, чтобы сделать такие языки еще жирнее, хотя эти усилия лучше было бы направить на поиск новых структур. Поэтому я попытался проанализировать некоторые из основных недостатков традиционных языков и показать, что от этих недостатков нельзя избавиться, если мы не найдем новый вид структуры языка.

В поиске альтернативы для традиционных языков нам нужно осознать, что система не может быть исторически чувствительной (допускать влияние выполнения одной программы на поведение следующей программы), если система не обладает некоторым состоянием (которое первая программа может изменять, а вторая может воспринимать). Итак, исторически чувствительная модель вычислительной системы должна обладать семантикой смены состояний хотя бы в этом слабом смысле. Но из этого не следует, что всякое вычисление должно существенно зависеть от сложного состояния, причем для каждой малой части вычислений требуется много изменений состояний (как в языках фон Неймана).

Чтобы проиллюстрировать некоторые альтернативы языкам фон Неймана, я предлагаю обозреть класс исторически чувствительных вычислительных систем, причем каждая система (а) характеризуется слабой зависимостью от семантики смены состояний, т. е. изменение состояния производится только один



раз в течение большого вычисления; (б) обладает просто структурированными состояниями и простыми правилами перехода; (в) существенно зависит от лежащей в ее основе аппликативной системы как в обеспечении базового языка программирования системы, так и в описании смены ее состояний.

Эти системы, которые я называю аппликативными системами переходов состояний (АСПС), описаны в разд. 14. Такие простые системы свободны от многих сложностей и слабостей языка фон Неймана и предусматривают мощный и обширный набор изменяемых частей. Однако они упоминаются только как грубые примеры из обширной области не-фон-неймановских систем с разнообразными притягательными свойствами. Я изучал эту область в течение последних трех или четырех лет и еще не нашел удовлетворительного решения для многих противоречивых требований, которым должен удовлетворять хороший язык. Однако я полагаю, что этот поиск указал полезный подход к проектированию не-фон-неймановских языков.

Данный подход включает четыре элемента, которые могут быть подытожены следующим образом.

(а) *Функциональный стиль программирования без переменных.* Описывается простая неформальная система функционального программирования (ФП). Она основывается на использовании комбинационных форм программ ФП. Приводится несколько программ для иллюстрации функционального программирования.

(б) *Алгебра функциональных программ.* Описывается алгебра, в которой переменные обозначают функциональные программы, а операции являются функциональными формами ФП, т. е. комбинационными формами программ ФП. Формулируются некоторые законы этой алгебры. Приводятся теоремы и примеры, которые показывают, как конкретные функциональные выражения могут преобразовываться в эквивалентные бесконечные разложения, объясняющие поведение функции. Алгебра ФП сравнивается с алгебрами, соответствующими классическим функциональным системам Чёрча и Карри.

(в) *Формальная система функционального программирования.* Описывается формальная система (ФФП), обобщающая возможности упомянутых выше неформальных систем ФП. Таким образом, система ФФП является точно определенной системой, обеспечивающей возможность использовать стиль функционального программирования систем ФП и их алгебру программ. Системы ФФП могут служить основой для аппликативных систем переходов состояний.

(г) *Аппликативные системы переходов состояний.* См. выше. В оставшейся части статьи описываются эти четыре элемента, и в заключение приводится резюме работы.

## 11. СИСТЕМЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ (СИСТЕМЫ ФП)

### 11.1. ВВЕДЕНИЕ

В этом разделе мы даем неформальное описание одного класса простых аппликативных систем программирования, называемых функциональными системами (ФП), в которых «программами» являются просто функции без переменных. Описание сопровождается некоторыми примерами и обсуждением различных свойств систем ФП.

В системе ФП находит применение фиксированное множество комбинационных форм, называемых функциональными формами. Они в сочетании с простыми описаниями являются единственными средствами построения новых функций из уже существующих. В них не используются переменные или правила подстановок, и они становятся операциями соответствующей алгебры программ. Все функции системы ФП относятся к одному типу: они отображают объекты на объекты и всегда работают с одним аргументом.

Напротив, в системе, основанной на лямбда-исчислении, с целью построения новых функций применяется лямбда-выражение с соответствующим множеством правил подстановок для переменных. Лямбда-выражение (с его правилами подстановок) в состоянии описать все возможные вычислимые функции всех мыслимых типов и с любым количеством аргументов. Эта свобода и выразительность чревата недостатками наряду с очевидными преимуществами. Она аналогична мощи неограниченных операторов управления в традиционных языках: вместе с неограниченной свободой приходит хаос. Если некто постоянно изобретает новые комбинационные формы применительно к возникающим ситуациям, как можно поступать в лямбда-исчислении, то он никогда не познакомится со стилем или полезными свойствами немногих комбинационных форм, подходящих ко всем случаям. Подобно тому как структурное программирование избегает многих управляющих операторов для получения программ с более простой структурой, лучшими свойствами и единообразными методами понимания их поведения, так и функциональное программирование избегает лямбда-выражений, подстановок и множественных типов функций. Благодаря этому получаются программы, построенные из хорошо знакомых функциональных форм с известными полезными свойствами. Эти программы структурированы таким образом, что часто можно понять и доказать их поведение механическим использованием алгебраических приемов, подобных тем, которые применяются при решении задач высшей алгебры.

В отличие от большинства конструкций программирования функциональные формы не нужно выбирать применительно к тому или иному случаю. Поскольку они являются операциями соответствующей алгебры, мы выбираем лишь такие функциональные формы, которые не только обеспечивают выразительные конструкции программирования, но и обладают привлекательными алгебраическими свойствами: они выбираются для максимизации силы и полезности алгебраических законов, связывающих их с другими функциональными формами системы.

В приводимом ниже описании мы не станем пунктуально различать (а) функциональный символ или выражение и (б) обозначаемую им функцию. Символы и выражения, служащие для обозначения функций, будут указываться посредством их использования. В разд. 13 описывается формальное обобщение систем ФП (системы ФФП); они могут служить для разъяснения любых неопределенностей относительно систем ФП.

## 11.2. ОПИСАНИЕ

Система ФП включает следующее:

- (1) множество  $O$  объектов;
- (2) множество  $F$  функций  $f$ , которые отображают объекты на объекты;
- (3) операцию, применение;
- (4) множество  $F$  функциональных форм; они служат для комбинирования существующих функций или объектов для формирования новых функций из  $F$ ;
- (5) множество  $D$  описаний, определяющих некоторые функции в  $F$  и присваивающих каждой из них имя.

Далее следует неформальное описание с примерами всех этих составных частей.

**11.2.1. Объекты,  $O$ .** Объект  $x$  является либо атомом, либо последовательностью  $(x_1, \dots, x_n)$ , элементы которой представляют собой объекты, либо  $\perp$  («основой», или «неопределенностью»). Таким образом, выбор множества  $A$  атомов определяет множество объектов. Мы будем считать, что  $A$  является множеством непустых строк из прописных букв, цифр и специальных символов, не используемых в нотации системы ФП. Некоторые из этих строк принадлежат к классу атомов, называемых «числами». Атом  $\emptyset$  служит для обозначения пустой последовательности и является единственным объектом, который представляет собой одновременно и атом, и последовательность. Атомы  $T$  и  $F$  служат для обозначения «истины» и «лжи».

На конструирование объектов накладывается одно важное ограничение: если  $x$  — это последовательность с элементом  $\perp$ ,

то  $x = \perp$ . Иначе говоря, «конструктор последовательностей» является «сохраняющим  $\perp$ ». Итак, ни одна последовательность в собственном смысле не содержит  $\perp$  в качестве элемента.

### Примеры объектов

$$\perp \quad 1.5 \quad \emptyset \quad AB3 \quad \langle AB, 1, 2, 3 \rangle \langle A, \langle B \rangle, C \rangle, D \rangle \quad \langle A, \perp \rangle = \perp$$

**11.2.2. Применение.** Система ФП включает единственную операцию, применение. Если  $f$  — это функция и  $x$  — объект, то  $f : x$  представляет собой *применение* и обозначает объект, являющийся результатом применения  $f$  к  $x$ . Здесь  $f$  — *оператор* применения, а  $x$  — *операнд*.

### Примеры применений

$$+ : \langle 1, 2 \rangle = 3 \quad ! : \langle A, B, C \rangle = \langle B, C \rangle \quad 1 : \langle A, B, C \rangle = A \quad 2 : \langle A, B, C \rangle = B$$

**11.2.3. Функции.** Все функции  $f$  из  $F$  отображают объекты на объекты и «сохраняют основу»:  $f : \perp = \perp$  для всех  $f$  из  $F$ . Любая функция из  $F$  является или *примитивной*, т. е. поставляемой вместе с системой, или *описываемой* (см. ниже), или *функциональной формой* (см. ниже).

Иногда бывает полезно различать два случая, в которых  $f : x = \perp$ . Если вычисление для  $f : x$  завершается и порождает объект  $\perp$ , мы говорим, что функция  $f$  *неопределена* в  $x$ , т. е.  $f$  завершается, но не дает содержательного значения в  $x$ . В противном случае мы говорим, что  $f$  *незавершила* в  $x$ .

**Примеры примитивных функций.** Наша цель состоит в том, чтобы обеспечить системы ФП широко применимыми и мощными примитивными функциями, а не слабыми примитивными функциями, которые затем могли бы быть использованы для описания полезных функций. В последующих примерах описываются некоторые типичные примитивные функции, многие из которых применяются в дальнейших примерах программ. В следующих описаниях мы употребляем вариант условного выражения Маккарти [17]; итак, мы пишем:

$$p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n; e_{n+1}.$$

вместо выражения Маккарти

$$(p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n, T \rightarrow e_{n+1}).$$

Следующие описания должны иметь место для всех объектов  $x, x_i, y, y_i, z, z_i$ .

### Функции селектора

$$1 : x \equiv x = \langle x_1, \dots, x_n \rangle \rightarrow x_1; \perp$$

и для любого положительного целого значения  $s$

$$s : x \equiv x = \langle x_1, \dots, x_n \rangle \& n \geq s \rightarrow x_s; \perp$$

Таким образом, например,  $3 : \langle A, B, C \rangle = C$  и  $2 : \langle A \rangle = \perp$ . Заметим, что символы функций 1, 2 и т. д. отличаются от атомов 1, 2 и т. д.

*Хвост (tail)*

$$tl : x \equiv x = \langle x_1 \rangle \rightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \& n \geq 2 \rightarrow \langle x_2, \dots, x_n \rangle; \perp$$

*Тождество (identity)*

$$id : x \equiv x$$

*Атом (atom)*

$$atom : x \equiv x \text{ является атомом} \rightarrow T; x \neq \perp \rightarrow F; \perp$$

*Равняется (equals)*

$$eq : x \equiv x = \langle y, z \rangle \& y = z \rightarrow T; x = \langle y, z \rangle \& y \neq z \rightarrow F; \perp$$

*Нуль (null)*

$$null : x \equiv x = \emptyset \rightarrow T; x \neq \perp \rightarrow F; \perp$$

*Обращение (reverse)*

$$reverse : x \equiv x = \emptyset \rightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \rightarrow \langle x_n, \dots, x_1 \rangle; \perp$$

*Дистрибутивно слева, дистрибутивно справа (distl, distr)*

$$distl : x \equiv x = \langle y, \emptyset \rangle \rightarrow \emptyset; x = \langle y, \langle z_1, \dots, z_n \rangle \rangle \rightarrow \langle y, z_1 \rangle, \dots$$

$$\dots, \langle y, z_n \rangle; \perp$$

$$distr : x \equiv x = \langle \emptyset, y \rangle \rightarrow \emptyset; x = \langle \langle y_1, \dots, y_n \rangle, z \rangle \rightarrow \langle y_1, z \rangle, \dots$$

$$\dots, \langle y_n, z \rangle; \perp$$

*Длина (length)*

$$length : x \equiv x = \langle x_1, \dots, x_n \rangle \rightarrow n; x = \emptyset \rightarrow 0; \perp$$

*Прибавить, вычесть, умножить и разделить (add, subtract, multiply and divide)*

$$+ : x \equiv x = \langle y, z \rangle \& y, z \text{ являются числами} \rightarrow y + z; \perp$$

$$- : x \equiv x = \langle y, z \rangle \& y, z \text{ являются числами} \rightarrow y - z; \perp$$

$$\times : x \equiv x = \langle y, z \rangle \& y, z \text{ являются числами} \rightarrow y \times z; \perp$$

$$\div : x \equiv x = \langle y, z \rangle \& y, z \text{ являются числами} \rightarrow y \div z; \perp \text{ (где } y \div 0 = \perp \text{)}$$

*Транспозиция (transpose)*

$$trans : x \equiv x = \langle \emptyset, \dots, \emptyset \rangle \rightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \rightarrow \langle y_1, \dots, y_m \rangle; \perp$$

где

$$x_i = \langle x_{i1}, \dots, x_{im} \rangle \text{ и } y_j = \langle x_{1j}, \dots, x_{nj} \rangle, 1 \leq i \leq n, 1 \leq j \leq m$$

и, или, отрицание (and, or, not)

$$\text{and} : x \equiv x = \langle T, T \rangle \rightarrow T; x = \langle T, F \rangle \vee x = \langle F, F \rangle \rightarrow F; \perp$$

и т. д.

*Присоединить слева, присоединить справа [append left; append right]*

$$\text{apndl} : x \equiv x = \langle y, \emptyset \rangle \rightarrow \langle y \rangle; x = \langle y, \langle z_1, \dots, z_n \rangle \rangle \rightarrow \langle y, z_1, \dots, z_n \rangle; \perp$$

$$\text{apndr} : x \equiv x = \langle \emptyset, z \rangle \rightarrow \langle z \rangle; x = \langle \langle y_1, \dots, y_n \rangle, z \rangle \rightarrow \langle y_1, \dots, y_n, z \rangle; \perp$$

*Правые селекторы, правый хвост (right selectors; right tail)*

$$1r : x \equiv x = \langle x_1, \dots, x_n \rangle \rightarrow x_n; \perp$$

$$2r : x \equiv x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow x_{n-1}; \perp$$

и т. д.

$$\text{tlr} : x \equiv x = \langle x_1 \rangle \rightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow \langle x_1, \dots, x_{n-1} \rangle; \perp$$

*Поворот влево, поворот вправо (rotate left; rotate right)*

$$\text{rotl} : x \equiv x = \emptyset \rightarrow \emptyset; x = \langle x_1 \rangle \rightarrow \langle x_1 \rangle;$$

$$x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow \langle x_2, \dots, x_n, x_1 \rangle; \perp$$

и т. д.

**11.2.4. Функциональные формы, F.** Функциональная форма представляет собой выражение, означающее функцию; такая функция зависит от функций или объектов, которые являются *параметрами* выражения. Так, например, если  $f$  и  $g$  — это любые функции, то  $f \circ g$  представляет собой функциональную форму, *композицию*  $f$  и  $g$ ;  $f$  и  $g$  являются ее параметрами, и она обозначает функцию, такую, что для любого объекта  $x$

$$(f \circ g) : x \equiv f : (g : x).$$

Для некоторых функциональных форм параметрами могут быть объекты. Например, для любого объекта  $x$ ,  $\bar{x}$  является функциональной формой, *константной* функцией от  $x$ , так что для любого объекта  $y$

$$\bar{x} : y \equiv y \perp \rightarrow \perp; x.$$

В частности,  $\bar{\perp}$  — это функция «всюду  $\perp$ ».

Ниже мы приводим некоторые функциональные формы, многие из которых используются позднее в этой статье. Мы применяем символы  $p$ ,  $f$  и  $g$  с индексами и без индексов для обозначения произвольных функций, а  $x$ ,  $x_1, \dots, x_n$ ,  $y$  являются произвольными объектами. Квадратные скобки [...] *служат*

для указания функциональной формы для *конструкции*, которая обозначает функцию, тогда как угловые скобки  $\langle \dots \rangle$  обозначают последовательности, которые являются объектами. Круглые скобки применяются и в конкретных функциональных формах (например, в *условии*), и в общем случае для указания группирования.

### Композиция

$$(f \circ g) : x \equiv g : (g : x)$$

### Конструкция

$$[f_1, \dots, f_n] : x \equiv \langle f_1 : x, \dots, f_n : x \rangle$$

(Напомним, что поскольку  $\langle \dots, \perp, \dots \rangle$  и все функции сохраняют  $\perp$ , то  $[f_1, \dots, f_n]$  тоже обладает этим свойством.)

### Условие

$$(p \rightarrow f; g) : x \equiv (p : x) = T \rightarrow f : x; (p : x) = F \rightarrow g : x; \perp$$

Условные *выражения* (используемые вне систем ФП для описания их функций) и условие *функциональной формы* идентифицируются знаком « $\rightarrow$ ». Это совсем разные, хотя и тесно связанные понятия, как показано в приведенных выше определениях. Но не должно возникать никакой путаницы, потому что все элементы условного выражения обозначают значения, тогда как все элементы условия функциональной формы обозначают функции, но заведомо не значения. Если не возникает никакой двусмысленности, мы опускаем ассоциирующиеся справа скобки; например, мы пишем

$$p_1 \rightarrow f_1; p_2 \rightarrow f_2; g \text{ вместо } (p_1 \rightarrow f_1; (p_2 \rightarrow f_2; g)).$$

*Константа* (Здесь  $x$  — это объектный параметр.)

$$\bar{x} : y \equiv y = \perp \rightarrow \perp; x$$

### Включение

$$/f : x \equiv x = \langle x_1 \rangle \rightarrow x_1;$$

$$x = \langle x_1, \dots, x_n \rangle \& n \geq 2 \rightarrow f : \langle x_1, /f : \langle x_2, \dots, x_n \rangle \rangle; \perp$$

Если  $f$  содержит единственный правый элемент  $u_f \neq \perp$ , где  $f : \langle x, u_f \rangle \in \{x, \perp\}$  для всех объектов  $x$ , то приведенное выше определение обобщается:  $/f : \emptyset = u_f$ . Итак,

$$/+; \langle 4, 5, 6 \rangle = + : \langle 4, + : \langle 5, /+ : \langle 6 \rangle \rangle \rangle = + : \langle 4, + : \langle 5, 6 \rangle \rangle = 15$$

$$/+ : \emptyset = 0$$

### Применить ко всем

$$\alpha f : x \equiv x = \emptyset \rightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f : x_1, \dots, f : x_n \rangle; \perp$$

Двоичное в единичное ( $x$  — объектный параметр)

$(\text{bu } f x) : y \equiv f : \langle x, y \rangle$

Таким образом,

$(\text{bu } + 1) : x \equiv 1 + x$

Пока (*while*)

$(\text{while } p \ f) : x \equiv p : x = T \rightarrow (\text{while } p \ f) : (f : x); p : x = F \rightarrow x; \perp$

Представленные выше функциональные формы обеспечивают эффективный метод вычисления значений обозначаемых ими функций (если они завершаются) при условии, что имеется возможность эффективно применять их параметры.

**11.2.5. Определения.** *Определением* в системе ФП является выражение вида

**Def**  $l = r$

где левая часть  $l$  представляет собой еще не использованный символ функции, а правая часть  $r$  является функциональной формой (которая может зависеть от  $l$ ). Оно выражает тот факт, что символ  $l$  должен обозначать функцию, задаваемую посредством  $r$ . Так, определение **Def**  $\text{last}l = 1 \cdot \text{reverse}$  определяет функцию  $\text{last}l$ , которая порождает последний элемент последовательности (или  $\perp$ ). Аналогично

**Def**  $\text{last} = \text{null} \cdot \text{tl} \rightarrow l; \text{last} \cdot \text{tl}$

описывает функцию  $\text{last}$ , такую же, как  $\text{last}l$ . Ниже подробно показано, как это описание можно было бы применить для вычисления  $\text{last} : \langle 1, 2 \rangle$ :

$\text{last} : \langle 1, 2 \rangle$	$=$
определение $\text{last}$	$\Rightarrow (\text{null} \cdot \text{tl} \rightarrow l; \text{last} \cdot \text{tl}) : \langle 1, 2 \rangle$
действие формы $(p \rightarrow f; g)$	$\Rightarrow \text{last} \cdot \text{tl} : \langle 1, 2 \rangle$
так как $\text{null} \cdot \text{tl} : \langle 1, 2 \rangle = \text{null} : \langle 2 \rangle = F$	
действие формы $f \cdot g$	$\Rightarrow \text{last} : (\text{tl} : \langle 1, 2 \rangle)$
определение примитивной хвостовой функции	$\Rightarrow \text{last} : \langle 2 \rangle$
определение $\text{last}$	$\Rightarrow (\text{null} \cdot \text{tl} \rightarrow l; \text{last} \cdot \text{tl}) : \langle 2 \rangle$
действие вида $(p \rightarrow f; g)$	$\Rightarrow l : \langle 2 \rangle$
так как $\text{null} \cdot \text{tl} : \langle 2 \rangle = \text{null} : \emptyset = T$	
определение селектора 1	$\Rightarrow 2$

Выше проиллюстрировано простое правило: для применения описанного символа замените его правой частью его определения. Разумеется, некоторые определения могут относиться к незавершаемым функциям. Множество  $D$  определений *корректно*, если никакие две левые части не совпадают.



**11.2.6. Семантика.** Из сказанного выше можно видеть, что система ФП определяется выбором следующих множеств: (а) Множество атомов  $A$  (которым определяется множество объектов). (б) Множество примитивных функций  $P$ . (в) Множество функциональных форм  $F$ . (г) Корректное множество определений  $D$ . Чтобы понять семантику такой системы, нужно знать, как вычислять  $f : x$  для любой функции  $f$  и любого объекта  $x$ , принадлежащих этой системе. Для  $f$  имеются четыре возможности:

- (1)  $f$  является примитивной функцией;
- (2)  $f$  является функциональной формой;
- (3) в  $D$  имеется одно определение,  $\text{Def } f \equiv r$ ;
- (4) ни один из предыдущих вариантов не имеет места.

Если  $f$  — примитивная функция, то мы располагаем ее определением и знаем, как ее применять. Если  $f$  — функциональная форма, то определение формы содержит информацию о том, как вычислять  $f : x$  в терминах параметров формы, что может быть сделано при дальнейшем использовании этих правил. Если функция  $f$  определена согласно  $\text{Def } f \equiv r$ , как в (3), то для отыскания  $f : x$  мы вычисляем  $r : x$ , а это можно сделать дальнейшим применением данных правил. Если ни один из рассмотренных вариантов не имеет места, то  $f : x \equiv \perp$ . Разумеется, работа по этим правилам может не завершиться при некоторых  $f$  и  $x$ ; в таком случае мы присваиваем значение  $f : x \equiv \perp$ .

### 11.3. ПРИМЕРЫ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ

В следующих примерах иллюстрируется функциональный стиль программирования. Поскольку этот стиль незнаком многим читателям, то в их восприятии на первых порах может возникнуть путаница. Важно помнить, что никакая часть определения функции не является результатом как таковым. Напротив, каждая часть представляет собой *функцию*, которую нужно применить к некоему аргументу, чтобы получить результат.

#### 11.3.1. Факториал

$\text{Def } ! \equiv \text{eq } 0 \rightarrow \bar{1}; \times \circ [\text{id}, ! \circ \text{subl}]$

где

$\text{Def } \text{eq } 0 \equiv \text{eq} \circ [\text{id}, \bar{0}]$

$\text{Def } \text{sub } 1 \equiv - \circ [\text{id}, \bar{1}]$

Здесь приводятся некоторые из промежуточных результатов, которые система ФП получила бы при вычислении  $! : 2$ :

$! : 2 \Rightarrow (\text{eq } 0 \rightarrow \bar{1}; \times \circ [\text{id}, ! \circ \text{subl}]) : 2 \Rightarrow \times \circ (\text{id}, ! \circ \text{subl}) : 2$

$$\begin{aligned} \Rightarrow X:\langle \text{id}:2, !\circ \text{sub } 1:2 \rangle &\Rightarrow X:\langle 2, !:1 \rangle \\ \Rightarrow X:\langle 2, X:\langle 1, !:0 \rangle \rangle \\ \Rightarrow X:\langle 2, X:\langle 1, \bar{I}:0 \rangle \rangle &\Rightarrow X:\langle 2, X:\langle 1, 1 \rangle \rangle \Rightarrow X:\langle 2, 1 \rangle \Rightarrow 2. \end{aligned}$$

В разд. 12 мы увидим, как теоремы алгебры программ ФП могут использоваться для доказательства того, что  $!$  является факториальной функцией.

**11.3.2. Внутреннее произведение (IP).** Мы уже видели ранее, как работает это описание.

$$\text{Def IP} \equiv (+) \circ (\alpha \times) \circ \text{trans}$$

**11.3.3. Умножение матриц (ММ).** Эта программа перемножения матриц порождает произведение любой пары  $(m, n)$  сопоставимых матриц, причем всякая матрица  $m$  представляется как последовательность ее строк:

$$m = \langle m_1, \dots, m_r \rangle,$$

где

$$m_i = \langle m_{i1}, \dots, m_{is} \rangle \text{ при } i=1, \dots, r.$$

Программа состоит из четырех этапов, читаемых справа налево; каждый из них, начиная с  $[1, \text{trans} \circ 2]$ , применяется по очереди к результату предыдущего этапа. Если аргумент — это  $\langle m, n \rangle$ , то первый этап дает

$$\langle m, n' \rangle,$$

где  $n' = \text{trans} : n$ . Вторым этапом дает

$$\langle m_1, n' \rangle, \dots, \langle m_r, n' \rangle,$$

где  $m_i$  — это строки из  $m$ . Третьим этапом  $\alpha \text{ distl}$  дает

$$\langle \text{distl} : \langle m_1, n' \rangle, \dots, \text{distl} : \langle m_r, n' \rangle \rangle = \langle p_1, \dots, p_r \rangle,$$

где

$$p_i = \text{distl} : \langle m_i, n' \rangle = \langle m_{i1}, n'_1 \rangle, \dots, \langle m_{is}, n'_s \rangle \text{ при } i=1, \dots, r$$

и  $n'_j$  — это  $j$ -й столбец из  $n$  ( $j$ -я строка из  $n'$ ). Таким образом,  $p_i$ , последовательность пар строк и столбцов, соответствует  $i$ -й строке произведения. Оператор  $\alpha \text{ IP}$ , или  $\alpha(\alpha \text{ IP})$ , вызывает применение  $\alpha \text{ IP}$  к каждой последовательности  $p_i$ , что, в свою очередь, влечет за собой применение  $\text{IP}$  к каждой паре строки и столбца в каждой  $p_i$ . Поэтому результатом последнего этапа является последовательность строк, составляющая произведение матриц. Если матрица не прямоугольная, или длина строки матрицы  $m$  отличается от длины столбца матрицы  $n$ , или если какой-то элемент из  $m$  или из  $n$  не является числом, то результат равен  $\perp$ .

Эта программа ММ не именует свои аргументы или какие-либо промежуточные результаты; она не содержит ни переменных, ни циклов, ни операторов управления, ни описаний процедур; в ней нет инструкций инициализации; по своей природе она не является пословной, она иерархически конструируется из более простых компонентов, использует универсальные формы и операторы ведения внутреннего хозяйства (в том числе  $\alpha f$ ,  $\text{distl}$ ,  $\text{distr}$ ,  $\text{trans}$ ); является совершенно общей; порождает  $\perp$  всякий раз, когда ее аргумент оказывается в каком-то смысле неприемлемым; не вносит необязательных ограничений на порядок вычисления (все применения к парам строк и столбцов могут выполняться параллельно или в любом порядке) и с использованием алгебраических законов (см. ниже) может быть преобразована в более «эффективные» или более «наглядные» программы (например, в рекурсивно описанную программу). Ни одним из этих свойств не обладает типичная программа фон Неймана для перемножения матриц.

Несмотря на свою непривычную, а поэтому озадачивающую форму, в отличие от большинства программ программа ММ описывает существенные операции умножения матриц без чрезмерно жесткого определения процесса или затемнения его частей. Поэтому из нее можно получать формальными преобразованиями много непосредственно выполнимых программ. Для компьютеров фон Неймана эта программа неэффективна по самой своей природе (в отношении использования памяти), но из нее можно вывести эффективные модификации, и можно представить себе реализации систем ФП, которые позволили бы выполнять ММ без расточительного использования памяти, предполагаемого приведенной здесь ее формой. Вопросы эффективности выходят за рамки проблематики этой статьи. Я позволю себе только заметить, что поскольку язык прост и не предписывает какой-либо привязки переменных лямбда-типа к данным, то, возможно, лучше было бы, если бы система оказалась способной выполнять некий вид «ленивого» вычисления [9, 10] и контролировать управление данными более эффективно, чем это удастся в системах, основанных на лямбда-исчислении.

#### 11.4. ЗАМЕЧАНИЯ О СИСТЕМАХ ФП

**11.4.1. Системы ФП как язык программирования.** Системы ФП столь скудны, что для некоторых читателей может оказаться затруднительной их интерпретация как языков программирования. При такой интерпретации функция  $f$  является программой, объект  $x$  представляет собой содержимое памяти, а  $f: x$  — это содержимое памяти после активации программы  $f$ , когда в памяти находится  $x$ . Множество определений представляет со-

бой библиотеку программ. Обеспечиваемые системой примитивные функции и функциональные формы являются базовыми операторами конкретного языка программирования. Итак, в зависимости от выбора примитивных функций и функциональных форм структура ФП обеспечивает большой класс языков с различными стилями и возможностями. Ассоциированная с каждым из этих языков алгебра программ зависит от конкретного набора функциональных форм. Представленные в этой статье примитивные функции, функциональные формы и программы заключают в себе попытку разработать один из таких возможных стилей.

**11.4.2. Ограничения систем ФП.** Системы ФП характеризуются рядом ограничений. Например, заданная система ФП представляет собой фиксированный язык, она не является исторически чувствительной: ни одна программа не в состоянии изменить библиотеку программ. Она способна интерпретировать входные и выходные данные только в том смысле, что  $x$  является входом, а  $f: x$  — это выход. Если имеется слабое множество примитивных функций и функциональных форм, то может оказаться, что удастся выразить не всякую вычислимую функцию.

Система ФП не может вычислять программу, потому что выражения функций не являются объектами. Нельзя также описывать новые функциональные формы в рамках системы ФП. (Оба этих ограничения устраняются в формальных системах функционального программирования (ФФП), в которых объекты «представляют» функции.) Так, ни одна система ФП не может включать функцию `apply`, такую, что

`apply: <x, y> == x : y`

потому что в левой части  $x$  — это объект, а в правой части  $x$  — это функция. (Заметим, что мы тщательно следили, чтобы множества символов функций и символов объектов были различными и непересекающимися; так,  $1$  является символом функции, а  $I$  — объектом.)

Основное ограничение системы ФП состоит в том, что они исторически нечувствительны. Поэтому их следует как-то расширить, прежде чем они смогут начать приносить практическую пользу. Обсуждение таких расширений приводится в разделах о системах ФФП и АСПС (разд. 13 и 14).

**11.4.3. Выразительная сила систем ФП.** Предположим, что две ФП-системы  $FP_1$  и  $FP_2$  обладают одинаковыми множествами объектов и одинаковыми множествами примитивных функций, но множество функциональных форм из  $FP_1$  включает соответствующее множество из  $FP_2$ , не совпадая с ним. Предпо-

ложим также, что обе системы могут выражать все вычислимые функции на объектах. Тем не менее мы можем сказать, что система  $FP_1$  более выразительна, чем система  $FP_2$ , поскольку всякое выражение функции в  $FP_2$  может быть продублировано в  $FP_1$ , но за счет использования функциональной формы, не принадлежащей к  $FP_2$ , система  $FP_1$  способна выражать некоторые функции более прямо и легко, чем система  $FP_2$ .

Я полагаю, что приведенные выше соображения могут быть развиты в теорию выразительной силы языков, согласно которой язык  $A$  оказался бы *более выразительным*, чем язык  $B$ , при следующих грубо сформулированных условиях. Во-первых, формируем все возможные функции любых типов на  $A$ , применяя все существующие функции к объектам и друг к другу всеми возможными способами до тех пор, пока нельзя будет сформировать никакую новую функцию какого-либо типа. (Множество объектов является типом. Множество непрерывных функций  $[T \rightarrow U]$  из типа  $T$  в тип  $U$  является типом. Если  $f \in [T \rightarrow U]$  и  $t \in T$ , то  $ft$  из  $U$  может быть сформировано применением  $f$  к  $t$ .) Делаем то же самое для языка  $B$ . Затем сравниваем любой тип из  $A$  с соответствующим типом из  $B$ . Если для любого типа верно, что тип из  $A$  включает соответствующий тип из  $B$ , то язык  $A$  более выразителен, чем язык  $B$  (или столь же выразителен). Если некоторый тип функции из  $A$  не имеет эквивалента в языке  $B$ <sup>1)</sup>, то языки  $A$  и  $B$  несравнимы по выразительной мощности.

**11.4.4. Преимущества системы ФП.** Системы ФП значительно проще, чем традиционные языки и языки, основанные на лямбда-исчислении, в основном по той причине, что в них употребляется только самая простая фиксированная система именования (именование функции в ее определении) с простым фиксированным правилом подстановки функции вместо ее имени. Поэтому в них отсутствует сложность как систем именования в традиционных языках, так и правил подстановок из лямбда-исчисления. Системы ФП допускают определение различных систем именования (см. разд. 13.3.4 и 14.7) для разных целей. От них не требуется сложность, поскольку многие программы могут работать совсем без них. Более существенно, что эти системы интерпретируют имена как функции, которые могут комбинироваться с другими функциями и не требуют специального подхода.

Системы ФП позволяют освободиться от традиционного по-

<sup>1)</sup> И наоборот, некий тип из  $B$  не имеет эквивалента в  $A$ . — *Прим. перев.*

словного программирования еще в большей степени, чем язык APL [12] (наиболее успешный на сегодняшний день подход к данной проблеме в рамках структуры фон Неймана), потому что они обеспечивают более мощный набор функциональных форм в едином мире выражений. Они позволяют развивать методы более высокого уровня, пригодные для размышлений над программами, манипулирования ими и их написания.

## 12. АЛГЕБРА ПРОГРАММ ДЛЯ СИСТЕМ ФП

### 12.1. ВВЕДЕНИЕ

Описываемая ниже алгебра программ является любительской работой в области алгебры, и я хочу показать, что любители могут успешно играть в эту игру и получать от нее удовлетворение, а также что эта игра не требует глубокого понимания логики и математики. Несмотря на свою простоту, она может помогать понимать и доказывать факты о программах систематическим, в каком-то смысле математическим способом.

Пока что доказательство корректности программ требует знания довольно сложных разделов математики и логики, свойств полных частично упорядоченных множеств, непрерывных функций, наименьших фиксированных точек функционалов, исчисления высказываний первого порядка, преобразователей предикатов, слабейших предусловий (здесь упомянуты лишь некоторые темы, необходимые для нескольких подходов к доказательству корректности программ). Эти темы оказывались очень полезными для профессионалов, которые сделали своим основным занятием изобретение методов доказательства; они опубликовали уйму блестящих работ по этой теме, начиная с исследований Маккарти и Флойда до более недавних работ Барстелла, Дейкстры, Манна и его сотрудников, Милнера, Морриса, Рейнольдса и многих других. Большая часть этих исследований основывается на фундаменте, заложенном Д. Скоттом (денотационные семантики) и Ч. Хоаром (аксиоматические семантики). Но по своему теоретическому уровню они недоступны большинству любителей, работающих вне этой специализированной области.

Если программисту средней квалификации требуется доказать корректность своей программы, ему понадобятся гораздо более простые методы, чем те, которые до сих пор развивали теоретики. Излагаемая ниже алгебра программ может послужить одной из отправных точек для такой дисциплины доказательства, и в сочетании с текущей работой по алгебраическим манипуляциям она может также способствовать построению основы для частичной автоматизации этой дисциплины.

Одно из преимуществ этой алгебры по сравнению с другими методами доказательств состоит в том, что программист может использовать свой язык программирования в качестве языка вывода доказательств, а не вынужден формулировать доказательства в отдельной логической системе, в которой можно только *рассуждать* о его программе.

Сердцевину алгебры программ составляют законы и теоремы, утверждающие, что одно выражение функции эквивалентно другому. Например, закон  $[f, g] \circ h \equiv [f \circ h, g \circ h]$  гласит, что конструкция из  $f$  и  $g$  (в композиции с  $h$ ) представляет собой такую же функцию, как конструкция из ( $f$  в композиции с  $h$ ) и ( $g$  в композиции с  $h$ ) вне зависимости от того, каковы функции  $f$ ,  $g$  и  $h$ . Такие законы легко понимаются, легко обосновываются, они мощны и удобны для практического применения. Однако мы хотим также применять такие законы для решений уравнений, в которых «неизвестная» функция появляется в обеих частях уравнения. Проблема состоит в том, что, если  $f$  удовлетворяет некоторому такому уравнению, часто случается, что и некое обобщение  $f'$  для  $f$  будет также удовлетворять тому же уравнению. Таким образом, для того чтобы обеспечить однозначность решений таких уравнений, мы должны потребовать, чтобы основания алгебры программ (с использованием введенного Скоттом понятия наименьших фиксированных точек непрерывных функционалов) гарантировали нам, что решения, получаемые алгебраическими манипуляциями, действительно являются наименьшими в этом смысле, а следовательно, единственными решениями.

Наша цель состоит в том, чтобы разработать основания алгебры программ, свободные от теоретических премудростей, и тем самым позволить программисту использовать простые алгебраические законы и одну или две теоремы из этих оснований для решения задач и построения доказательств точно таким же механическим образом, как мы решаем алгебраические задачи в высшей школе, при этом ничего не зная о наименьших фиксированных точках или преобразователях предикатов.

Возникает одна конкретная проблема, относящаяся к этим основаниям: если заданы уравнения вида

$$f \equiv p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; E_1(f) \quad (1)$$

где  $p_i$  и  $q_i$  — это функции, не включающие  $f$ , а  $E_1(f)$  — функциональное выражение, включающее  $f$ , то законы алгебры часто позволяют формально «расширить» это уравнение еще на одну «фразу» за счет вывода

$$E_1(f) \equiv p_{i+1} \rightarrow q_{i+1}; E_{i+1}(f) \quad (2)$$

что при замене  $E_1(f)$  в (1) на правую часть из (2) порождает

$$\bar{f} \equiv p_0 \rightarrow q_0; \dots; p_{i+1} \rightarrow q_{i+1}; E_{i+1}(f) \quad (3)$$

Такое формальное расширение может продолжаться без ограничений. В таком случае из оснований должен следовать ответ на вопрос: когда наименьшая  $\bar{f}$ , удовлетворяющая (1), может быть представлена бесконечным разложением

$$\bar{f} \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (4)$$

в котором последняя фраза, включающая  $\bar{f}$ , была опущена, так что теперь мы имеем решение с правой частью, свободной от  $\bar{f}$ ? Такие решения полезны в двух отношениях: во-первых, они дают доказательства «завершимости» в том смысле, что вследствие (4)  $\bar{f}:x$  определено тогда и только тогда, когда существует значение  $n$ , при котором для всякого  $i < n$  имеем  $p_i: x = F$  и  $p_n: x = T$ , и определено  $q_n: x$ . Во-вторых, формула (4) дает последовательное описание функции  $\bar{f}$ , которое часто может прояснить поведение функции.

Приводимые в последующем разделе основания алгебры — это скромный шаг в сторону сформулированной выше цели. Для ограниченного класса уравнений соответствующая «теорема линейного расширения» дает полезный ответ относительно того, когда можно перейти от неопределенно продолжаемых уравнений типа (1) к бесконечным разложениям типа (4). Для более обширного класса уравнений более общая «теорема разложения» дает менее полезный ответ на сходные вопросы. К счастью, можно найти более общие теоремы, покрывающие дополнительные классы уравнений. Но в настоящее время достаточно знать только следствия этих двух простых фундаментальных теорем, чтобы понимать теоремы и примеры, представленные в этом разделе.

Результаты подраздела об основаниях обобщаются в отдельном, более раннем подразделе, озаглавленном «теоремы разложения», без ссылок на концепции фиксированных точек. Сам подраздел об основаниях помещен позже и может быть опущен читателями, не желающими углубляться в этот предмет.

## 12.2. ЗАКОНЫ АЛГЕБРЫ ПРОГРАММ

В алгебре программ для системы ФП переменные варьируются в пределах множества функций системы. «Операциями» алгебры являются функциональные формы системы. Так, например,  $[f, g] \cdot h$  — это выражение такой алгебры для описанной выше системы ФП, в которой переменные  $f$ ,  $g$  и  $h$  обозначают произвольные функции этой системы. И



$$\{f, g\} \circ h \equiv [f \circ h, g \circ h]$$

представляет собой закон алгебры, который гласит, что, какие бы функции не были выбраны для  $f$ ,  $g$  и  $h$ , функция в левой части не отличается от функции в правой части. Таким образом, этот алгебраический закон только формулирует по-иному следующее утверждение относительно любой системы ФП, которая включает функциональные формы  $[f, g]$  и  $f \circ g$ .

**Утверждение.** Для всяких функций  $f$  и  $g$  и для любых объектов  $x$  верно  $([f, g] \circ h) : x \equiv [f \circ h, g \circ h] : x$ .

*Доказательство.*

$$\begin{aligned} ([f, g] \circ h) : x &= [f, g] : (h : x) \text{ по определению композиции} \\ &= \langle f : (h : x), g : (h : x) \rangle \text{ по определению конструкции} \\ &= \langle (f \circ h) : x, (g \circ h) : x \rangle \text{ по определению композиции} \\ &= [f \circ h, g \circ h] : x \text{ по определению конструкции } \square \end{aligned}$$

Области действия некоторых законов меньше, чем область всех объектов. В частности,  $1 \circ [f, g] \equiv f$  не верно для объектов  $x$ , таких, что  $g : x = \perp$ . Мы записываем

$$\text{defined} \circ g \rightarrow \rightarrow 1 \circ [f, g] \equiv f$$

чтобы указать, что закон (или теорема) справа справедлив в области объектов  $x$ , для которых  $\text{defined} \circ g : x = T$ , где

$$\text{Def defined} \equiv \bar{T}$$

т. е.  $\text{defined} : x \equiv x = \perp \rightarrow \perp$ ;  $T$ . В общем случае мы будем записывать *ограниченное функциональное уравнение*

$$p \rightarrow \rightarrow f \equiv g$$

которое означает, что для любого объекта  $x$  всякий раз, когда  $p : x = T$ , верно  $f : x = g : x$ .

Обычная алгебра имеет дело с двумя операциями, сложением и вычитанием, и нуждается в немногих законах. Алгебра программ имеет дело с большим числом операций (функциональных форм) и поэтому нуждается в большем количестве законов.

Каждый из следующих законов требует, чтобы выполнялось соответствующее утверждение. Интересующийся читатель легко найдет большинство доказательств таких утверждений (два из них приведены ниже). Определим сначала обычное упорядочение функций и эквивалентность в терминах этого упорядочения:

**Определение.**  $f \leq g$  тогда и только тогда, когда для всех объектов  $x$  либо  $f : x = \perp$ , либо  $f : x = g : x$

**Определение.**  $f \equiv g$  тогда и только тогда, когда  $f \leq g$  и  $g \leq f$ .

Легко убедиться в том, что  $\leq$  является частичным упорядочением;  $f \leq g$  означает, что  $g$  является обобщением для  $f$ , а  $f \equiv g$  тогда и только тогда, когда  $f: x = g: x$  для всяких объектов  $x$ . Приведем теперь список алгебраических законов, перечисляемых по парам основных функциональных форм.

## I. Композиция и конструкция

- I.1  $[f_1, \dots, f_n] \circ g = [f_1 \circ g, \dots, f_n \circ g]$   
 I.2  $\alpha f \circ [g_1, \dots, g_n] = [f \circ g_1, \dots, f \circ g_n]$   
 I.3  $[f_0 [g_1, \dots, g_n]] = f_0 [g_1, [f_0 [g_2, \dots, g_n]]]$  при  $n \geq 2$ .  
 $\quad \quad \quad = f_0 [g_1, f_0 [g_2, \dots, f_0 [g_{n-1}, g_n] \dots]]$   
 $[f \circ [g]] = g$   
 I.4  $f \circ [\bar{x}, g] = (\text{bu } f \ x) \circ g$   
 I.5  $1 \circ [f_1, \dots, f_n] \leq f_1$   
 $s \circ [f_1, \dots, f_s, \dots, f_n] \leq f_s$  для любого селектора  $s$ ,  $s \leq n$   
 $\text{defined} \circ f_i$  (при всех  $i \neq s$ ,  $1 \leq i \leq n$ )  $\rightarrow \rightarrow s \circ [f_1, \dots, f_n] = f_s$   
 I.5.1  $[f_1 \circ 1, \dots, f_n \circ n] \circ [g_1, \dots, g_n] = [f_1 \circ g_1, \dots, f_n \circ g_n]$   
 I.6  $\text{tl} \circ [f_1] \leq \emptyset$  и  $\text{tl} \circ [f_1, \dots, f_n] \leq [f_2, \dots, f_n]$  при  $n \geq 2$   
 $\text{defined} \circ f_1 \rightarrow \rightarrow \text{tl} \circ [f_1] = \emptyset$   
 и  $\text{tl} \circ [f_1, \dots, f_n] = [f_2, \dots, f_n]$  при  $n \geq 2$   
 I.7  $\text{distl} \circ [f, [g_1, \dots, g_n]] = [[f, g_1], \dots, [f, g_n]]$ .  
 $\text{defined} \circ f \rightarrow \rightarrow \text{distl} \circ [f, \emptyset] = \emptyset$

Аналогичный закон справедлив для дистрибутивности справа (distr).

- I.8  $\text{apndl} \circ [f, [g_1, \dots, g_n]] = [f, g_1, \dots, g_n]$   
 $\text{null} \circ g \rightarrow \rightarrow \text{apndl} \circ [f, g] = [f]$

и так далее, для  $\text{apndr}$ , обращения (reverse), поворота (rotl) и др.

- I.9  $[\dots, \bar{1}, \dots] = \bar{1}$   
 I.10  $\text{apndl} \circ [f \circ g, \alpha f \circ h] = \alpha f \circ \text{apndl} \circ [g, h]$   
 I.11  $\text{pair} \ \& \ \text{not} \circ \text{null} \circ 1 \rightarrow \rightarrow \text{apndl} \circ [[1 \circ 1, 2], \text{distr} \circ [\text{tl} \circ 1, 2]] = \text{distr}$   
 где  $f \ \& \ g = \text{and} \circ [f, g]$ ;  $\text{pair} = \text{atom} \rightarrow \bar{F}$ ;  $\text{eq} \circ \text{length}, \bar{2}$

**II. Композиция и условие** (ассоциирующие справа скобки опущены) (Закон II.2 указан в [16, р. 493].)

- II.1  $(p \rightarrow f; g) \circ h = p \circ h \rightarrow f \circ h; g \circ h$   
 II.2  $h \circ (p \rightarrow f; g) = p \rightarrow h \circ f; h \circ g$   
 II.3  $\text{or} \circ [q, \text{not} \circ q] \rightarrow \rightarrow \text{and} \circ [p, q] \rightarrow f; \text{and} \circ [p, \text{not} \circ q] \rightarrow g; h$   
 $= p \rightarrow (q \rightarrow f; g); h$   
 II.3.1  $p \rightarrow (p \rightarrow f; g); h = p \rightarrow f; h$

## III. Композиция и смешение

$$\text{III.1} \quad \bar{x} \circ f \leq \bar{x} \\ \text{defined} \circ f \rightarrow \rightarrow \bar{x} \circ f = \bar{x}$$

$$\text{III.1.1} \quad \bar{1} \circ f = f \circ \bar{1} = \bar{1}$$

$$\text{III.2} \quad f \circ \text{id} = \text{id} \circ f = f$$

$$\text{III.3} \quad \text{pair} \rightarrow \rightarrow 1 \circ \text{distr} = [1 \circ 1, 2] \text{ также: } \text{pair} \rightarrow \rightarrow 1 \circ t = 2 \text{ и т. д.}$$

$$\text{III.4} \quad \alpha(f \circ g) = \alpha f \circ \alpha g$$

$$\text{III.5} \quad \text{null} \circ g \rightarrow \rightarrow \alpha f \circ g = \bar{\emptyset}$$

#### IV. Условие и конструкция

$$\text{IV.1} \quad [f_1, \dots, (p \rightarrow g; h), \dots, f_n] \\ = p \rightarrow [f_1, \dots, g, \dots, f_n]; [f_1, \dots, h, \dots, f_n]$$

$$\text{IV.1.1} \quad [f_1, \dots, (p_1 \rightarrow g_1, \dots, p_n \rightarrow g_n; h), \dots, f_m] \\ = p_1 \rightarrow [f_1, \dots, g_1, \dots, f_m]; \\ \dots; p_n \rightarrow [f_1, \dots, g_n, \dots, f_m]; [f_1, \dots, h, \dots, f_m]$$

Этим завершается данный список алгебраических законов; он никоим образом не является исчерпывающим; существуют многие другие законы.

#### Доказательства двух законов

Мы приводим доказательства справедливости утверждения для законов I.10 и I.11, которые немного сложнее, чем большинство остальных.

#### Утверждение 1

$$\text{apndl} \circ [f \circ g, \alpha f \circ h] = \alpha f \circ \text{apndl} \circ [g, h]$$

*Доказательство.* Мы покажем, что для любого объекта  $x$  обе указанные выше функции порождают одинаковые результаты.

*Случай 1.*  $h : x$  — это не последовательность и не  $\emptyset$ . Тогда обе части при применении к  $x$  порождают  $\perp$ .

*Случай 2.*  $h : x = \emptyset$ . Тогда

$$\begin{aligned} \text{apndl} \circ [f \circ g, \alpha f \circ h] : x &= \text{apndl} : \langle f \circ g : x, \emptyset \rangle = \langle f : (g : x) \rangle \\ \alpha f \circ \text{apndl} \circ [g, h] : x &= \alpha f \circ \text{apndl} : \langle g : x, \emptyset \rangle = \alpha f : \langle g : x \rangle \\ &= \langle f : (g : x) \rangle \end{aligned}$$

*Случай 3.*  $h : x = \langle y_1, \dots, y_n \rangle$ . Тогда

$$\begin{aligned} \text{apndl} \circ [f \circ g, \alpha f \circ h] : x &= \text{apndl} : \langle f \circ g : x, \alpha f : \langle y_1, \dots, y_n \rangle \rangle \\ &= \langle f : (g : x), f : y_1, \dots, f : y_n \rangle \\ \alpha f \circ \text{apndl} \circ [g, h] : x &= \alpha f \circ \text{apndl} : \langle g : x, \langle y_1, \dots, y_n \rangle \rangle \\ &= \alpha f : \langle g : x, y_1, \dots, y_n \rangle \\ &= \langle f : (g : x), f : y_1, \dots, f : y_n \rangle \end{aligned}$$

#### Утверждение 2

$$\text{Pair} \circ \cdot \text{not} \circ \text{null} \circ 1 \rightarrow \rightarrow \text{apndl} \circ [[1^2, 2], \text{distr} \circ [t1 \circ 1, 2]]] = \text{distr},$$

где  $f \& g$  является функцией:  $\text{and} \circ [f, g]$ ,  $\text{and} f^2 = f \circ f$ .

**Доказательство.** Мы покажем, что обе части порождают одинаковый результат при их применении к любой паре  $\langle x, y \rangle$ , где  $x \neq \emptyset$  согласно сформулированному ограничению.

**Случай 1.**  $x$  является атомом или  $\perp$ . Тогда  $\text{distr} : \langle x, y \rangle = \perp$ , потому что  $x \neq \emptyset$ . Левая часть тоже порождает  $\perp$  при применении к  $\langle x, y \rangle$ , поскольку  $\text{tl} \circ 1 : \langle x, y \rangle = \perp$  и все функции сохраняют  $\perp$ .

**Случай 2.**  $x = \langle x_1, \dots, x_n \rangle$ . Тогда

$$\begin{aligned} \text{apndl} \circ [1^2, 2], \text{distr} \circ [\text{tl} \circ 1, 2]] : \langle x, y \rangle \\ &= \text{apndl} : \langle 1 : x, y \rangle, \text{distr} : \langle \text{tl} : x, y \rangle \\ &= \text{apndl} : \langle x_1, y \rangle, \emptyset \rangle = \langle x_1, y \rangle \text{ если } \text{tl} : x = \emptyset \\ &= \text{apndl} : \langle x_1, y \rangle, \langle x_2, y \rangle, \dots, \langle x_n, y \rangle \rangle \text{ если } \text{tl} : x \neq \emptyset \\ &= \langle x_1, y \rangle, \dots, \langle x_n, y \rangle \\ &= \text{distr} : \langle x, y \rangle. \end{aligned}$$

□

### 12.3. ПРИМЕР: ЭКВИВАЛЕНТНОСТЬ ДВУХ ПРОГРАММ УМНОЖЕНИЯ МАТРИЦ

Мы рассматривали ранее программу умножения матриц:

**Def**  $\text{MM} = \alpha \alpha \text{ IP} \circ \alpha \text{ distr} \circ [1, \text{trans} \circ 2]$ .

Теперь мы покажем, что ее начальный сегмент  $\text{MM}'$ , где

**Def**  $\text{MM}' = \alpha \alpha \text{ IP} \circ \text{distl} \circ \text{distr}$

может быть описан рекурсивно. ( $\text{MM}'$  «перемножает» пару матриц после того, как вторая матрица транспонирована. Заметим, что  $\text{MM}'$  в отличие от  $\text{MM}$  дает  $\perp$  для любых аргументов, которые не образуют пар.) Иначе говоря, мы покажем, что  $\text{MM}'$  удовлетворяет следующему уравнению, рекурсивно описывающему ту же функцию (на парах):

$f = \text{null} \circ 1 \rightarrow \overline{\emptyset}$ ;  $\text{apndl} \circ [\alpha \text{ IP} \circ \text{distl} \circ [1 \circ 1, 2], f \circ [\text{tl} \circ 1, 2]]$ .

Наше доказательство примет форму демонстрации того, что следующая функция  $R$ , где

**Def**  $R = \text{null} \circ 1 \rightarrow \overline{\emptyset}$ ;  $\text{apndl} \circ [\alpha \text{ IP} \circ \text{distl} \circ [1 \circ 1, 2], \text{MM}' \circ [\text{tl} \circ 1, 2]]$

при любых парах  $\langle x, y \rangle$  эквивалентна функции  $\text{MM}'$ . Функция  $R$  «умножает» две матрицы, когда первая матрица содержит более чем 0 строк; вычисляется первая строка «произведения» (при помощи  $\alpha \text{ IP} \circ \text{distl} \circ [1 \circ 1, 2]$ ) и присоединяется к «произведению» остатка первой матрицы и второй матрицы. Таким образом, нам нужна теорема

$\text{pair} \rightarrow \rightarrow \text{MM} \equiv R$ ,

а из этого сразу следует

$$MM \equiv MM' \circ [1, \text{trans} \circ 2] \equiv R \circ [1, \text{trans} \circ 2], \text{ где}$$

$$\text{Def pair} \equiv \text{atom} \rightarrow \bar{F}; \text{eq} \circ [\text{length}, \bar{2}].$$

**Теорема:**  $\text{pair} \rightarrow \rightarrow MM' = R$ ,

где

$$\text{Def } MM' \equiv \alpha \alpha \text{ IP} \circ \alpha \text{ distl} \circ \text{distr}$$

$$\text{Def } R \equiv \text{null} \circ 1 \rightarrow \bar{\emptyset}; \text{apndl} \circ [\alpha \text{ IP} \circ \text{distl} \circ [1^2, 2], MM' \circ [\text{tl} \circ 1, 2]]$$

*Доказательство.*

*Случай 1.*  $\text{pair} \& \text{null} \circ 1 \rightarrow \rightarrow MM' \equiv R$

$$\text{pair} \& \text{null} \circ 1 \rightarrow \rightarrow R \equiv \emptyset \text{ по определению } R$$

$$\text{pair} \& \text{null} \circ 1 \rightarrow \rightarrow MM' \equiv \emptyset$$

так как  $\text{distr} : \langle \emptyset, x \rangle = \emptyset$  по определению  $\text{distr}$

и  $\alpha f : \emptyset = \emptyset$  по определению «Применить ко всем»

Таким образом,  $\alpha \alpha \text{ IP} \circ \alpha \text{ distl} \circ \text{distr} : \langle \emptyset, x \rangle = \emptyset$ .

Поэтому  $\text{pair} \& \text{null} \circ 1 \rightarrow \rightarrow MM' \equiv R$ .

*Случай 2.*  $\text{pair} \& \text{not} \circ \text{null} \circ 1 \rightarrow \rightarrow MM' \equiv R$ .

$$\text{pair} \& \text{not} \circ \text{null} \circ 1 \rightarrow \rightarrow R \equiv R' \text{ по определению } R \text{ и } R', \text{ где} \quad (1)$$

$$\text{Def } R' \equiv \text{apndl} \circ [\alpha \text{ IP} \circ \text{distl} \circ [1^2, 2], MM \circ [\text{tl} \circ 1, 2]].$$

Заметим, что

$$R' \equiv \text{apndl} \circ [f \circ g, \alpha f \circ h],$$

где

$$f \equiv \alpha \text{ IP} \circ \text{distl}$$

$$g \equiv [1^2, 2]$$

$$h \equiv \text{distr} \circ [\text{tl} \circ 1, 2]$$

$$\alpha f \equiv \alpha (\alpha \text{ IP} \circ \text{distl}) \equiv \alpha \alpha \text{ IP} \circ \alpha \text{ distl} \text{ (согласно III.4).} \quad (2)$$

Таким образом, согласно 1.10,

$$R' \equiv \alpha f \circ \text{apndl} \circ [g, h]. \quad (3)$$

Теперь  $\text{apndl} \circ [g, h] \equiv \text{apndl} \circ [[1^2, 2], \text{distr} \circ [\text{tl} \circ 1, 2]]$ , поэтому в силу 1.11

$$\text{pair} \& \text{not} \circ \text{null} \circ 1 \rightarrow \rightarrow \text{apndl} \circ [g, h] \equiv \text{distr}. \quad (4)$$

Итак, мы имеем, согласно (1), (2), (3) и (4),

$$\begin{aligned} \text{pair} \& \text{not} \circ \text{null} \circ 1 \rightarrow \rightarrow R &\equiv R' \\ &\equiv \alpha f \circ \text{distr} \equiv \alpha \alpha \text{ IP} \circ \alpha \text{ distl} \circ \text{distr} \equiv MM'. \end{aligned}$$

Случаи 1 и 2, взятые вместе, доказывают теорему.  $\square$

## 12.4. ТЕОРЕМЫ РАЗЛОЖЕНИЙ

В следующих подразделах мы займемся «решением» некоторых простых уравнений (здесь под «решением» мы понимаем отыскание «наименьшей» функции, которая удовлетворяет уравнению). Для этого нам понадобятся следующие понятия и результаты, выводимые из дальнейшего подраздела об основаниях алгебры, где появляются их доказательства.

**12.4.1. Разложение.** Предположим, что у нас имеется уравнение вида

$$f \equiv E(f) \quad (E1)$$

где  $E(f)$  — выражение, включающее  $f$ . Предположим далее, что имеется бесконечная последовательность функций  $f_i$  для  $i=0, 1, 2, \dots$ , каждая из которых имеет следующий вид:

$$f_0 \equiv \perp \\ f_{i+1} \equiv p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; \perp \quad (E2)$$

где  $p_i'$  и  $q_i'$  — это конкретные функции, так что  $E$  обладает свойством

$$E(f_i) \equiv f_{i+1} \text{ при } i=0, 1, 2, \dots \quad (E3)$$

Тогда мы говорим, что выражение  $E$  разложимо и имеет  $f_i$  в качестве *аппроксимирующих функций*.

Если  $E$  разложимо и обладает аппроксимирующими функциями, как в (E2), и если  $f$  является решением для (E1), то  $f$  можно записать как бесконечное разложение

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (E4)$$

означающее, что для любого  $x$   $f: x \neq \perp$  тогда и только тогда, когда имеется значение  $n \geq 0$ , такое, что (а)  $p_i: x = F$  для всех  $i < n$ , (б)  $p_n: x = T$ , (в)  $q_n: x \neq \perp$ . Если  $f: x \neq \perp$ , то  $f: x = q_n: x$  для этого значения  $n$ . (Сказанное выше является следствием «теоремы о разложении».)

**12.4.2. Линейное разложение.** Существует более удобное средство решения некоторых уравнений, оно применимо, когда для любой функции  $h$

$$E(h) \equiv p_0 \rightarrow q_0; E_1(h) \quad (LE1)$$

и существуют  $p_i$  и  $q_i$ , такие, что

$$E_1(p_i \rightarrow q_i; h) \equiv p_{i+1} \rightarrow q_{i+1}; E_1(h) \text{ при } i=0, 1, 2, \dots \quad (LE2)$$

$$\text{и } E_1(\perp) \equiv \perp. \quad (LE3)$$

При этих условиях выражение  $E$  называется *линейно разложимым*.

Если оно таково и  $f$  является решением для

$$f \equiv E(f) \quad (\text{LE4})$$

то  $E$  и  $f$  можно переписать как бесконечное разложение

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (\text{LE5})$$

с применением  $p_i$  и  $q_i$ , генерируемых по (LE1) и (LE2).

Хотя  $p_i$  и  $q_i$  из (E4) и (LE5) не являются единственными для данной функции, может оказаться возможным найти дополнительные ограничения, которые придают им однозначность, и в таком случае разложение (LE5) представляло бы собой каноническую форму функции  $f$ . Даже без однозначности такие разложения часто позволяют доказывать эквивалентность двух различных функциональных выражений и часто проясняют поведение функции.

## 12.5. ТЕОРЕМА О РЕКУРСИИ

Используя три из сформулированных выше законов и линейное разложение, можно доказать следующую довольно общую теорему, которая дает разложение, проясняющее многие рекурсивно определенные функции.

**Теорема рекурсии.** Пусть  $f$  является решением для

$$f \equiv p \rightarrow g; Q(f) \quad (1)$$

где

$$Q(k) \equiv h \circ [i, k \circ j] \text{ для любой функции } k \quad (2)$$

и  $p, g, h, i, j$  — это любые заданные функции; тогда

$$f \equiv p \rightarrow g, p \circ j \rightarrow Q(g); \dots; p \circ j^n \rightarrow Q^n(g); \dots \quad (3)$$

(где  $Q^n(g)$  есть  $h \circ [i, Q^{n-1}(g) \circ j]$  и  $j^n$  есть  $j \circ j^{n-1}$  для  $n \geq 2$ ) и

$$Q^n(g) \equiv h \circ [i, i \circ j, \dots, i \circ j^{n-1}, g \circ j^n]. \quad (4)$$

**Доказательство.** Мы доказываем, что  $p \rightarrow g; Q(f)$  линейно разложимо. Пусть  $p_n, q_n$  и  $k$  суть произвольные функции. Тогда

$$\begin{aligned} Q(p_n \rightarrow q_n; k) &\equiv h \circ [i, (p_n \rightarrow q_n; k) \circ j] && \text{по (2)} \\ &\equiv h \circ [i, (p_n \circ j \rightarrow q_n \circ j; k \circ j)] && \text{по II.1} \\ &\equiv h \circ (p_n \circ j \rightarrow [i, q_n \circ j]; [i, k \circ j]) && \text{по IV.1 (5)} \\ &\equiv p_n \circ j \rightarrow h \circ [i, q_n \circ j]; h \circ [i, k \circ j] && \text{по II.2} \\ &\equiv p_n \circ j \rightarrow Q(q_n); Q(k) && \text{по (2)} \end{aligned}$$

Итак, если  $p_0 \equiv p$  и  $q_0 \equiv q$ , то (5) дает  $p_1 \equiv p \circ j$  и  $q_1 \equiv Q(g)$ , а в общем случае дает следующие функции, удовлетворяющие

(LE2):

$$p_n \equiv p \circ j^n \text{ и } q_n \equiv Q^n(g). \quad (6)$$

Наконец,

$$\begin{aligned} Q(\overline{1}) &\equiv h \circ [i, \overline{1} \circ j] \\ &\equiv h \circ [i, \overline{1}] && \text{по III.1.1} \\ &\equiv h \circ \overline{1} && \text{по I.9} \\ &\equiv \overline{1} && \text{по III.1.1} \end{aligned} \quad (7)$$

Итак, (5) и (6) обосновывают (LE2), и (7) обосновывает (LE3) при  $E_1 \equiv Q$ . Если принять  $E(f) \equiv f \rightarrow g$ ;  $Q(f)$ , то мы получаем (LE1); таким образом, выражение  $E$  линейно разложимо. Поскольку  $f$  является решением для  $f \equiv E(f)$ , вывод (3) следует из (6) и (LE5). Теперь

$$\begin{aligned} Q^n(g) &\equiv h \circ [i, Q^{n-1}(g) \circ j] \\ &\equiv h \circ [i, h \circ [i \circ j, \dots, h \circ [i \circ j^{n-1}, g \circ j^n] \dots]] \text{ по I.1, примененно-} \\ &\quad \text{му многократно} \\ &\equiv h \circ [i, i \circ j, \dots, i \circ j^{n-1}, g \circ j^n] \text{ по I.3.} \end{aligned} \quad (8)$$

Результат (8) является вторым выводом (4).

**12.5.1. Пример: доказательство корректности для рекурсивной факториальной функции.** Пусть  $f$  — это решение для

$$f \equiv \text{eq } 0 \rightarrow \overline{1}; \times \circ [\text{id}, f \circ s]$$

где

$$\text{Def } s \equiv - \circ [\text{id}, \overline{1}] \text{ (вычитание 1).}$$

Тогда  $f$  удовлетворяет предположению теоремы рекурсии для  $p \equiv \text{eq } 0$ ,  $g \equiv \overline{1}$ ,  $h \equiv \times$ ,  $i \equiv \text{id}$  и  $j \equiv s$ .

Поэтому

$$f \equiv \text{eq } 0 \rightarrow \overline{1}; \dots; \text{eq } 0 \circ s^n \rightarrow Q^n(\overline{1}); \dots$$

и

$$Q^n(\overline{1}) \equiv / \times \circ [\text{id}, \text{id} \circ s, \dots, \text{id} \circ s^{n-1}, \overline{1} \circ s^n].$$

Теперь  $\text{id} \circ s^k \equiv s^k$  согласно III.2 и  $\text{eq } 0 \circ s^n \rightarrow \overline{1} \circ s^n \equiv \overline{1}$  согласно III.1, поскольку  $\text{eq } 0 \circ s^n : x$  следует  $\text{defined} \circ s^n : x$ , а также

$$\text{eq } 0 \circ s^n : x \equiv \text{eq } 0 : (x - n) \equiv x = n.$$

Итак, если  $\text{eq } 0 \circ s^n : x = T$ , то  $x = n$  и

$$Q^n(\overline{1}) : n = n \times (n-1) \times \dots \times (n - (n-1)) \times (\overline{1} : (n-n)) = n!$$



Используя эти результаты для  $\overline{I \circ s^n}$ ,  $\text{eq } 0 \circ s^n$  и  $Q^n(1)$  в предыдущем разложении для  $f$ , получаем

$$f: x \equiv x = 0 \rightarrow \overline{I}; \dots; x = n \rightarrow n \times (n-1) \times \dots \times 1 \times 1; \dots$$

Таким образом, мы доказали, что  $f$  завершается в точности на множестве неотрицательных целых, и поэтому она является факториальной функцией.

## 12.6. ТЕОРЕМА ОБ ИТЕРАЦИИ

В сущности, это дополнение к теореме о рекурсии. Она дает простое разложение для многих итеративных программ.

**Теорема об итерации.** Пусть  $f$  — это решение (т. е. наименьшее решение) для  $f \equiv p \rightarrow g; h \circ f \circ k$ ; тогда

$$f \equiv p \rightarrow g; p \circ k \rightarrow h \circ g \circ k; \dots; p \circ k^n \rightarrow h^n \circ g \circ k^n; \dots$$

*Доказательство.* Пусть  $h' \equiv h \circ 2$ ,  $i' \equiv \text{id}$ ,  $j' \equiv k$ , тогда

$$f \equiv p \rightarrow g; h' \circ [i', f \circ j']$$

поскольку  $h \circ 2 \circ [\text{id}, f \circ k] \equiv h \circ f \circ k$  согласно I.5 ( $\text{id}$  определен всюду, кроме  $\perp$ , а уравнение справедливо для  $\perp$ ). Итак, теорема о рекурсии дает

$$f \equiv p \rightarrow g; \dots; p \circ k^n \rightarrow Q^n(g); \dots$$

где

$$\begin{aligned} Q^n(g) &\equiv h \circ 2 \circ [\text{id}, Q^{n-1}(g) \circ k] \\ &\equiv h \circ Q^{n-1}(g) \circ k \equiv h^n \circ g \circ k^n \quad \text{по I.5} \end{aligned}$$

□

**12.6.1. Пример: доказательство корректности для итеративной факториальной функции.** Пусть  $f$  — решение для

$$f \equiv \text{eq } 0 \circ 1 \rightarrow 2; f \circ [s \circ 1, \times]$$

где  $\text{Def } s \equiv - \circ [\text{id}, \overline{I}]$  (вычитание 1). Мы хотим доказать, что  $f: \langle x, 1 \rangle = x!$  тогда и только тогда, когда  $x$  — это неотрицательное целое. Пусть  $p \equiv \text{eq } 1$ ,  $g \equiv 2$ ,  $h \equiv \text{id}$ ,  $k \equiv [s \circ 1, \times]$ . Тогда  $f \equiv p \rightarrow g$ ,  $h \circ f \circ k$  и поэтому

$$f \equiv p \rightarrow g; \dots; p \circ k \rightarrow g \circ k^n, \dots \quad (1)$$

в силу теоремы об итерации, так как  $h^n \equiv \text{id}$ .

Мы хотим показать, что

$$\text{pair} \rightarrow \rightarrow k^n \equiv [a_n, b_n] \quad (2)$$

верно для любого значения  $n \geq 1$ , где

$$a_n \equiv s^n \circ 1 \quad (3)$$

$$b_n \equiv / \times \circ [s^{n-1} \circ 1, \dots, s \circ 1, 1, 2]. \quad (4)$$

Теперь (2) имеет место для  $n \equiv 1$  по определению  $k$ . Предположим, что (2) верно для некоторого значения  $n \geq 1$ , и докажем, что это утверждение верно для  $n+1$ . Имеем

$$\text{pair} \rightarrow \rightarrow k^{n+1} \equiv k \circ k^n \equiv [s \circ 1, \times] \circ [a_n, b_n] \quad (5)$$

поскольку (2) справедливо для  $n$ . И поэтому

$$\text{pair} \rightarrow \rightarrow k^{n+1} \equiv [s \circ a_n \times \circ [a_n, b_n]] \text{ согласно I.1 и I.5.} \quad (6)$$

Для перехода от (5) к (6) нужно проверить, что когда  $a_n$  или  $b_n$  порождает  $\perp$  в (5), то же самое имеет место в (6). Далее,

$$s \circ a_n \equiv s^{n+1} \circ 1 \equiv a_{n+1} \quad (7)$$

$$\begin{aligned} \times \circ [a_n, b_n] &\equiv / \times \circ [s^n \circ 1, \dots, s \circ 1, 1, 2] \\ &\equiv b_{n+1} \text{ согласно I.3} \end{aligned} \quad (8)$$

Сочетание (6), (7) и (8) дает

$$\text{pair} \rightarrow \rightarrow k^{n+1} \equiv [a_{n+1}, b_{n+1}]. \quad (9)$$

Итак, отношение (2) справедливо для  $n=1$ , а также выполняется при  $n+1$ , если оно верно при  $n$ ; поэтому по индукции оно справедливо при любом  $n \geq 1$ . Теперь (2) дает для пар

$$\begin{aligned} \text{defined} \circ k^n \rightarrow \rightarrow p \circ k^n &\equiv \text{eq } 0 \circ 1 \circ [a_n, b_n] \\ &\equiv \text{eq } 0 \circ a_n \equiv \text{eq } 0 \circ s^n \circ 1 \end{aligned} \quad (10)$$

$$\begin{aligned} \text{defined} \circ k^n \rightarrow \rightarrow g \circ k^n &\equiv 2 \circ [a_n, b_n] \\ &\equiv / \times \circ [s^{n-1} \circ 1, \dots, s \circ 1, 1, 2] \end{aligned} \quad (11)$$

(в обоих случаях используется I.5). Далее из (1) следует, что  $f: \langle x, 1 \rangle$  определено тогда и только тогда, когда имеется такое значение  $n$ , для которого  $p \circ k^i: \langle x, 1 \rangle = F$  для всех  $i < n$  и  $p \circ k^n: \langle x, 1 \rangle = T$ , т. е., согласно (10),  $\text{eq } 0 \circ s^n: x = T$ , т. е.  $x = n$ ; и  $g \circ k^n: \langle x, 1 \rangle$  определено; в таком случае, согласно (11),

$$f: \langle x, 1 \rangle = / \times: \langle 1, 2, \dots, x-1, x, 1 \rangle = n! \quad \square$$

а именно это нам требуется доказать.

**12.6.2. Пример: доказательство эквивалентности двух итеративных программ.** В этом примере мы хотим показать, что две итеративно описанные программы  $f$  и  $g$  реализуют одну и ту же функцию. Пусть  $f$  — это решение для

$$f \equiv p \circ 1 \rightarrow 2; \quad h \circ f \circ [k \circ 1, 2]. \quad (1)$$

Пусть  $g$  — решение для

$$g \equiv p \circ 1 \rightarrow 2; \quad g \circ [k \circ 1, h \circ 2]. \quad (2)$$

Тогда по теореме об итерации

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (3)$$

$$g \equiv p_0' \rightarrow q_0'; \dots; p_n' \rightarrow q_n'; \dots \quad (4)$$

где (при  $r^0 \equiv \text{id}$  для любого значения  $r$ ) для  $n=0, 1, \dots$

$$p_n \equiv p \circ 1 \circ [k \circ 1, 2]^n \equiv p \circ 1 \circ [k^{n \circ 1}, 2] \text{ по I.5.1} \quad (5)$$

$$q_n \equiv h^{n \circ 2} \circ [k \circ 1, 2]^n \equiv h^{n \circ 2} \circ [k^{n \circ 1}, 2] \text{ по I.5.1} \quad (6)$$

$$p_n' \equiv p \circ 1, \quad h \circ [k \circ 1, 2]^n \equiv p \circ 1 \circ [k^{n \circ 1}, h^{n \circ 2}] \text{ по I.5.1} \quad (7)$$

$$q_n' \equiv 2 \circ [k \circ 1, h \circ 2]^n \equiv 2 \circ [k^{n \circ 1}, h^{n \circ 2}] \text{ по I.5.1} \quad (8)$$

Теперь, воспользовавшись I.5, получаем из сказанного

$$\text{defined} \circ 2 \rightarrow \rightarrow p_n \equiv p \circ k^{n \circ 1} \quad (9)$$

$$\text{defined} \circ h^{n \circ 2} \rightarrow \rightarrow p_n' \equiv p \circ k^{n \circ 1} \quad (10)$$

$$\text{defined} \circ k^{n \circ 1} \rightarrow \rightarrow q_n \equiv q_n' \equiv h^{n \circ 2} \quad (11)$$

Итак,

$$\text{defined} \circ h^{n \circ 2} \rightarrow \rightarrow \text{defined} \circ 2 \equiv \bar{T} \quad (12)$$

$$\text{defined} \circ h^{n \circ 2} \rightarrow \rightarrow p_n \equiv p_n' \quad (13)$$

и

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow h^{n \circ 2}; \dots \quad (14)$$

$$g \equiv p_0' \rightarrow q_0'; \dots; p_n' \rightarrow h^{n \circ 2}; \dots \quad (15)$$

поскольку из  $p_n$  и  $p_n'$  следует нужное ограничение для  $q_n \equiv q_n' \equiv h^{n \circ 2}$ .

Теперь предположим, что существует такое выражение  $x$ , при котором  $f \circ x \neq g \circ x$ . Тогда найдется значение  $n$ , такое, что  $p_i' \circ x = p_i' \circ x = F$  для  $i < n$  и  $p_n \circ x \neq p_n' \circ x$ . Согласно (12) и (13), такое может случиться, только если  $h^{n \circ 2} \circ x = \perp$ . Но так как  $h$  сохраняет  $\perp$ , то  $h^{m \circ 2} \circ x = \perp$  для всех  $m \geq n$ . Поэтому  $f \circ x = g \circ x = \perp$  в силу (14) и (15). Это противоречит предположению, что существует выражение  $x$ , при котором  $f \circ x \neq g \circ x$ . Поэтому  $f \equiv g$ .

Этот пример (принадлежащий Дж. Моррису) более изящно рассматривается в [16]. Однако есть основания считать, что наша интерпретация более конструктивна, более автоматически приводит к постановке ключевых вопросов и обеспечивает лучшее понимание поведения обеих функций.

## 12.7. НЕЛИНЕЙНЫЕ УРАВНЕНИЯ

Прежние примеры касались «линейных» уравнений (в которых «неизвестная» функция не имеет аргумента, зависящего от нее самой). Остается открытым вопрос о существовании простых разложений, которые решали бы квадратные уравнения и уравнения более высокого порядка.

В предыдущих примерах рассматривались решения для  $f \equiv \equiv E(f)$ , где выражение  $E$  линейно разложимо. В следующем примере фигурирует  $E(f)$ , которое квадратично и разложимо (но не линейно разложимо).

**12.7.1. Пример: доказательство идемпотентности [16].** Пусть  $f$  — это решение для

$$f \equiv E(f) \equiv p \rightarrow \text{id}; f_2 \circ h. \quad (1)$$

Мы хотим доказать, что  $f \equiv f^2$ . Проверяем, что выражение  $E$  разложимо (разд. 12.4.1) при следующих аппроксимирующих функциях:

$$f_0 \equiv \perp \quad (2a)$$

$$f_n \equiv p \rightarrow \text{id}; \dots; p \circ h^{n-1} \rightarrow h^{n-1}; \perp \text{ при } n > 0. \quad (26)$$

Заметим сначала, что  $p \rightarrow \rightarrow f_n \equiv \text{id}$  и поэтому

$$p \circ h^1 \rightarrow \rightarrow f_n \circ h^1 \equiv h^1. \quad (3)$$

$$\text{Теперь } E(f) \equiv p \rightarrow \text{id}; \perp^2 \circ h \equiv f_1 \quad (4)$$

и

$$E(f_n)$$

$$\equiv p \rightarrow \text{id}; f_n \circ (p \rightarrow \text{id}; \dots; p \circ h^{n-1} \rightarrow h^{n-1}; \perp) \circ h$$

$$\equiv p \rightarrow \text{id}; f_n \circ (p \circ h \rightarrow h; \dots; p \circ h^n \rightarrow h^n; \perp \circ h)$$

$$\equiv p \rightarrow \text{id}; p \circ h \rightarrow f_n \circ h; \dots; p \circ h^n \rightarrow f_n \circ h^n, f_n \circ \perp$$

$$\equiv p \rightarrow \text{id}; p \circ h \rightarrow h; \dots; p \circ h^n \rightarrow h^n; \perp \text{ by (3)}$$

$$\equiv f_{n+1} \quad (5)$$

Итак, выражение  $E$  разложимо согласно (4) и (5); поэтому в силу (2) и разд. 12.4.1 ( $E4$ )

$$f \equiv p \rightarrow \text{id}; \dots; p \circ h^n \rightarrow h^n, \dots \quad (6)$$

Но, по теореме об итерации, (6) дает

$$f \equiv p \rightarrow \text{id}; f \circ h. \quad (7)$$

Теперь, если  $p : x = T$ , то  $f : x = x = f^2 : x$  в силу (1). Если  $p : x = F$ , то

$$\begin{aligned} f : x &= f^2 \circ h : x && \text{по (1)} \\ &= f : (f \circ h : x) = f : (f : x) && \text{по (7)} \\ &= f^2 : x. \end{aligned}$$

Если  $p : x$  не является ни  $T$ , ни  $F$ , то  $f : x = \perp = f^2 : x$ . Таким образом,  $f \equiv f^2$ .

## 12.8. ОСНОВАНИЯ ДЛЯ АЛГЕБРЫ ПРОГРАММ

В этом разделе мы преследуем цель установить справедливость результатов, сформулированных в разд. 12.4. Последующие разделы не зависят от этого материала, и поэтому читатели могут при желании пропустить его. Мы пользуемся стандартными понятиями и результатами из [16], но для объектов и функций применяется нотация, принятая в этой статье.

В качестве области определения (и области значений) для всех функций выбирается множество  $O$  объектов (включая  $\perp$ ) данной системы ФП. Будем обозначать через  $F$  множество функций, а через  $F$  — множество функциональных форм такой системы ФП. Обозначаем как  $E(f)$  любое функциональное выражение, включающее функциональные формы, примитивные и выведенные функции, а также символ функции  $f$ , и будем относиться к  $E$  как к функционалу, который отображает функцию  $f$  на соответствующую функцию  $E(f)$ . Предполагаем, что все  $f \in F$  сохраняют  $\perp$  и что все функциональные формы из  $F$  соответствуют непрерывным функционалам по каждому переменному (например,  $[f, g]$  непрерывна как по  $f$ , так и по  $g$ ). (Все примитивные функции представленной ранее системы ФП сохраняют  $\perp$ , а все ее функциональные формы непрерывны.)

**Определение.** Пусть  $E(f)$  — это функциональное выражение. Пусть

$$f_0 \equiv \perp$$

$$f_{i+1} \equiv p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; \perp \text{ при } i=0, 1, \dots$$

где  $p_i, q_i \in F$ . Пусть  $E$  обладает тем свойством, что

$$E(f_i) \equiv f_{i+1} \text{ при } i=0, 1, \dots$$

Тогда выражение  $E$  называется *разложимым с аппроксимирующими функциями*  $f_i$ . Мы пишем

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots$$

чтобы указать, что  $f \equiv \lim_i \{f_i\}$ , где  $f_i$  имеет приведенный выше вид. Правую часть мы называем *бесконечным разложением* для  $f$ . Будем считать, что  $f : x$  определено тогда и только тогда, когда существует значение  $n \geq 0$ , такое, что (а)  $p_1 : x = F$  для всех  $i < n$  и (б)  $p_n : x = T$  и (в)  $q_n : x$  определено; в таком случае  $f : x = q_n : x$ .

**Теорема о разложении.** Пусть выражение  $E(f)$  разложимо с указанными выше аппроксимирующим функциями. Пусть  $f$  — наименьшая функция, удовлетворяющая

$$f = E(f).$$

Тогда

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots$$

**Доказательство.** Поскольку  $E$  — это композиция непрерывных функционалов (из  $F$ ), включающих только монотонные функции (сохраняющие  $\perp$  функции из  $F$ ) в качестве константных членов, то функция  $E$  непрерывна [16]. Поэтому ее наименьшая фиксированная точка  $f$  имеет вид  $\lim_i \{E^i(\perp)\} \equiv \lim_i \{f_i\}$ , а по определению это и есть указанное выше бесконечное разложение для  $f$ .

**Определение.** Пусть  $E(f)$  — функциональное выражение, удовлетворяющее

$$E(h) \equiv p_0 \rightarrow q_0; E_1(h) \text{ при всех } h \in F, \quad (\text{LE1})$$

причем существуют  $p_i \in F$  и  $q_i \in F$ , такие, что

$$E_1(p_i \rightarrow q_i; h) \equiv p_{i+1} \rightarrow q_{i+1}; E_1(h) \text{ при всех } h \in F \text{ и } i=0, 1, \dots \quad (\text{LE2})$$

и

$$E_1(\perp) \equiv \perp. \quad (\text{LE3})$$

Тогда выражение  $E$  называется *линейно разложимым* по этим  $p_i$  и  $q_i$ .

**Теорема о линейном разложении.** Пусть выражение  $E$  линейно разложимо по  $p_i$  и  $q_i$ ,  $i=0, 1, \dots$ . Тогда выражение  $E$  разложимо с аппроксимирующими функциями

$$f_0 \equiv \perp \quad (1)$$

$$f_{i+1} \equiv p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; \perp. \quad (2)$$

**Доказательство.** Мы хотим показать, что  $E(f_i) \equiv f_{i+1}$  для любого значения  $i \geq 0$ . Теперь

$$E(f_0) \equiv p_0 \rightarrow q_0; E_1(\perp) \equiv p_0 \rightarrow q_0; \perp \equiv f_1 \text{ по (LE1) (LE3) (1)}. \quad (3)$$

Пусть фиксировано значение  $i > 0$  и пусть

$$f_i \equiv p_0 \rightarrow q_0; w_1 \quad (4a)$$

$$w_1 \equiv p_1 \rightarrow q_1; w_2 \text{ и т. д.} \quad (4б)$$

$$w_{i-1} \equiv p_{i-1} \rightarrow q_{i-1}; \perp. \quad (4-)$$

Тогда для этого решения  $i > 0$

$$E(f_i) \equiv p_0 \rightarrow q_0; E_1(f_i) \text{ по (LE1)} \quad (\text{LE1})$$

$$E_1(f_i) \equiv p_1 \rightarrow q_1; E_1(w_1) \text{ по (LE2) и (4a)}$$

$$E_1(w_1) \equiv p_2 \rightarrow q_2; E_1(w_2) \text{ по (LE2) и (4б) и т. д.}$$

$$E_1(w_{i-1}) \equiv p_i \rightarrow q_i; E_1(\overline{\perp}) \text{ по (LE2) и (4-)} \\ \equiv p_i \rightarrow q_i; \perp \text{ по (LE3)}$$

Сочетание этих отношений дает

$$E(f_i) \equiv f_{i+1} \text{ для произвольного } i > 0 \text{ по (2)}. \quad (5)$$

В силу (3) отношение (5) также справедливо при  $i \geq 0$ , поэтому оно верно для всех значений  $i \geq 0$ . Следовательно, выражение  $E$  разложимо и обладает требуемыми аппроксимирующими функциями.

**Следствие.** Если выражение  $E$  линейно разложимо по  $p_i$  и  $q_i$  при  $i = 0, 1, \dots$  и  $f$  — наименьшая функция, удовлетворяющая

$$f \equiv E(f) \quad (\text{LE4})$$

то

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (\text{LE5})$$

## 12.9. АЛГЕБРА ПРОГРАММ ДЛЯ ЛЯМБДА-ИСЧИСЛЕНИЯ И ДЛЯ КОМБИНАТОРОВ

Поскольку лямбда-исчисление Чёрча [5] и система комбинаторов, разработанная Шёнфинкелем и Карри [6], являются в первую очередь математическими системами для представления понятия применения функций и поскольку они более мощны, чем системы ФП, то естественно заняться исследованием того, как будет выглядеть основанная на этих системах алгебра программ. В лямбда-исчислении и комбинаторах эквиваленты для композиции  $f \circ g$  из ФП имеют вид

$$\lambda f g x. (f(gx)) \equiv B$$

где  $B$  — простой комбинатор, определенный Карри. В системах Чёрча или Карри отсутствуют подходящие непосредственные эквиваленты для объекта  $\langle x, y \rangle$  из ФП, однако, согласно Ландину [14] и Бёрджу [4], можно воспользоваться примитивными функциями префикс, голова, хвост, нуль и атом, чтобы ввести понятие списковых структур, которые соответствуют последовательностям из ФП. Тогда применив понятие из ФП для списка, можно сделать эквивалентом конструкции ФП выражение  $\lambda f g x. \langle f x, g x \rangle$  из лямбда-исчисления. Эквивалентом в комбинаторах является выражение, включающее префикс, нулевой список и два или более основных комбинатора. Оно столь сложно, что я не пытаюсь привести его здесь.

При использовании лямбда-исчисления или комбинаторных выражений для функциональных форм  $f \circ g$  и  $[f, g]$  для пред-

ставления закона I.1 из алгебры ФП,  $[f, g] \circ h \equiv [f \circ h, g \circ h]$ , результатом является настолько сложное выражение, что смысл закона затемняется. Единственный способ сделать этот смысл ясным в любой системе состоит в том, чтобы вывести наименования для двух функционалов: композиция  $\equiv B$  и конструкция  $\equiv A$ , так что  $Bfg \equiv f \circ g$  и  $Afg \equiv [f, g]$ .

Тогда I.1 принимает вид

$$B(Afg)h \equiv A(Bfh)(Bgh)$$

который все еще не столь прозрачен, как закон из ФП.

Суть сказанного состоит в том, что если некто хочет сформулировать ясные законы в системах Чёрча или Карри наподобие соответствующих законов в алгебре ФП, то он сталкивается с необходимостью выбрать определенные функционалы (например, композицию и конструкцию) в качестве основных операций алгебры и либо присвоить им короткие имена, либо, что предпочтительнее, представить их некоторой специальной нотацией, как в ФП. Если сделать это и обеспечить примитивы, объекты, списки и т. д., результатом будет ФП-подобная система, в которой не появляются обычные лямбда-выражения или комбинаторы. Даже тогда эти версии Чёрча или Карри для системы ФП в силу меньшей ограниченности представляют некоторые проблемы, которые не возникают в системах ФП:

(а) Версии Чёрча и Карри предоставляют функции многих типов и позволяют описывать функции, которые не существуют в системах ФП. Так, функция  $Bf$  не имеет эквивалента в системах ФП. Эта дополнительная мощь влечет за собой проблемы совместимости типов. Например, для функции  $f \circ g$  включается ли диапазон значений  $g$  в область определения  $f$ ? В системах ФП все функции обладают одними и теми же областями определения и значений.

(б) Семантика лямбда-исчисления Чёрча зависит от правил подстановок, которые просто формулируются, но следствия которых с большим трудом поддаются осознанию. Далеко не все понимают истинную сложность этих правил, но ее очевидным свидетельством является преуспевание логиков, публиковавших «доказательства» теоремы Чёрча — Россера, в которых упускались из виду те или иные из этих сложностей. (Теорема Чёрча — Россера или доказательство Скотта существования модели [22] требуются для того, чтобы показать, что у лямбда-исчисления имеется непротиворечивая семантика.) Описание чистого Лиспа длительное время содержало связанную с этим ошибку (проблема «funarg»). Аналогичные проблемы возникают и в системе Карри.

Напротив, формальная (ФФП) версия систем ФП (описываемая в следующем разделе) не содержит переменных и



включает только одно элементарное правило подстановки (функция вместо ее имени), и для нее существование непротиворечивой семантики может быть доказано относительно просто в духе рассуждений, развитых Д. Скоттом, а также Манной и др. [16]. Такое доказательство приведено в [18].

## 12.10. ЗАМЕЧАНИЯ

Намеченная выше алгебра программ потребует большой работы для распространения ее на более обширные классы или уравнения и для обобщения ее законов и теорем за рамки приведенных здесь элементарных случаев. Было бы интересно исследовать алгебру для ФП-подобной системы, в которой конструктор последовательности не сохраняет  $\perp$  (закон 1.5 становится более сильным, но теряется IV.1). Другие интересные проблемы состоят в том, чтобы (а) найти правила обеспечения однозначности разложений, задав канонические формы для функций; (б) найти алгоритмы разложения и анализа поведения функций для различных классов аргументов и (в) исследовать способы использования законов и теорем алгебры в качестве основных правил либо для схемы формального, умозрительного «ленивого вычисления» [9, 10], либо для схемы, работающей в процессе выполнения алгоритма. В таких схемах, например, закон  $1 \circ [f, g] \leq f$  служил бы для того, чтобы избежать вычисления  $g : x$ .

## 13. ФОРМАЛЬНЫЕ СИСТЕМЫ ДЛЯ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ (СИСТЕМЫ ФФП)

### 13.1. ВВЕДЕНИЕ

Как мы уже видели, система ФП включает множество функций, которое зависит от ее множества примитивных функций, ее множества функциональных форм и ее множества определений. В частности, ее множество функциональных форм фиксировано раз навсегда, и это множество решающим образом определяет мощь системы. Например, если ее множество функциональных форм пусто, то ее множество собственно функций в точности совпадает с множеством примитивных функций. В системах ФФП можно создавать новые функциональные формы. Функциональные формы представляются последовательностями объектов; первый элемент последовательности определяет, какую форму она представляет, тогда как остальные элементы являются параметрами формы.

Возможность описывать в системах ФФП новые функциональные формы является одним из следствий принципиального

различия между этими системами и системами ФП: в системах ФФП объекты служат для «представления» функций систематическим образом. В других отношениях ФФП почти не отличаются от ФП. Они похожи на языки сведения (Red) из ранней публикации [2], но проще этих языков.

Сначала мы приведем простой синтаксис систем ФФП, затем неформально обсудим их семантику, рассмотрев примеры, и, наконец, введем их формальную семантику.

## 13.2. СИНТАКСИС

Мы описываем множество  $O$  объектов и множество  $E$  выражений системы ФФП. Они зависят от выбора некоего множества  $A$  атомов, которое мы принимаем как заданное. Мы предполагаем, что множеству  $A$  принадлежат  $T$  (истина),  $F$  (ложь),  $\emptyset$  (пустая последовательность), а также «числа» различных видов и т. д.

- (1) Основа  $\perp$  является объектом, но не атомом.
- (2) Всякий атом является объектом.
- (3) Всякий объект является выражением.
- (4) Если  $x_1, \dots, x_n$  — это объекты (выражения), то  $\langle x_1, \dots, x_n \rangle$  является объектом (соответственно выражением), называемым последовательностью (длины  $n$ ) при  $n \geq 1$ . Объект (выражение)  $x_i$  при  $1 \leq i \leq n$  является элементом последовательности  $\langle x_1, \dots, x_i, \dots, x_n \rangle$  ( $\emptyset$  одновременно является и последовательностью, и атомом; длина этой последовательности равна 0).

(5) Если  $x$  и  $y$  — это выражения, то  $(x : y)$  является выражением, называемым применением. Здесь  $x$  — оператор, а  $y$  — операнд. И  $x$ , и  $y$  представляют собой элементы выражения.

(6) Если  $x = \langle x_1, \dots, x_n \rangle$  и если один из элементов  $x$  есть  $\perp$ , то  $x = \perp$ . Иначе говоря,  $\langle \dots, \perp, \dots \rangle = \perp$ .

(7) Все объекты и выражения формируются использованием приведенных выше правил конечное число раз.

Подвыражением выражения  $x$  является либо само  $x$ , либо подвыражение некоего элемента из  $x$ . Объект ФФП представляет собой выражение, не содержащее применения в качестве своего подвыражения. Если задано одно и то же множество атомов, то объекты ФФП и ФП одинаковы.

## 13.3. НЕФОРМАЛЬНЫЕ ЗАМЕЧАНИЯ О СЕМАНТИКЕ ФФП

**13.3.1. Значение выражения; семантическая функция  $\mu$ .** Всякое выражение  $e$  из ФФП имеет значение  $\mu e$ , которое всегда является объектом;  $\mu e$  отыскивается повторяемой заменой всякого самого внутреннего применения в  $e$  на его значение. Если этот процесс не завершается, то значение  $e$  есть  $\perp$ . Значением

самого внутреннего применения  $(x : y)$  (поскольку оно самое внутреннее, то  $x$  и  $y$  должны быть объектами) является результатом применения к  $y$  функции, *представляемой*  $x$ , точно так же, как в системах ФП, с той разницей, что в системах ФФП функции представляются объектами, а не функциональными выражениями, причем атомы (а не символы функций) представляют примитивные и определяемые функции, а последовательно представляют функции ФП, обозначаемые функциональными формами.

Связь между объектами и представляемыми ими функциями задается *функцией представления*  $\rho$  системы ФФП. (И  $\rho$ , и  $\mu$  принадлежат описанию системы, но не самой системе.) Итак, если атом  $NULL$  представляет функцию  $null$  системы ФП, то  $\rho NULL = null$  и значение  $(NULL : A)$  есть  $\mu(NULL : A) = (\rho NULL) : A = null : A = F$ .

Здесь, как и выше, мы используем двоеточие в двух смыслах. Когда оно находится между двумя объектами, как в  $(NULL : A)$ , оно идентифицирует применение ФФП, которое обозначает только себя; когда же оно находится между *функцией* и объектом, как в  $(\rho NULL) : A$  или в  $null : A$ , оно идентифицирует ФП-подобное применение, которое обозначает *результат* применения функции к объекту.

Тот факт, что операторы ФФП являются объектами, делает возможным существование функции  $apply$ , которая не имеет смысла в системах ФП:

$apply : \langle x, y \rangle = (x : y)$

Результат  $apply : \langle x, y \rangle$ , а именно  $(x : y)$ , не имеет смысла в системах ФП, причем это верно на двух уровнях. Во-первых, сочетание  $(x : y)$  само по себе не является объектом; оно иллюстрирует другое различие между системами ФП и ФФП: некоторые функции из ФФП, например  $apply$ , отображают объекты на выражения, а не непосредственно на объекты (в отличие от функций из ФП). Однако *значение*  $apply : \langle x, y \rangle$  представляет собой объект (см. ниже). Во-вторых,  $\langle x : y \rangle$  не может быть даже промежуточным результатом в системе ФП; это сочетание не имеет смысла в системах ФП, потому что  $x$  — это объект, а не функция, а системы ФП не ассоциируют функции с объектами. Теперь если  $APPLY$  представляет  $apply$ , то значение  $(APPLY : \langle NULL, A \rangle)$  равно

$$\begin{aligned} \mu(APPLY : \langle NULL, A \rangle) &= \mu((\rho APPLY) : \langle NULL, A \rangle) \\ &= \mu(apply : \langle NULL, A \rangle) \\ &= \mu(NULL : A) = \mu((\rho NULL) : A) \\ &= \mu(null : A) = \mu F = F. \end{aligned}$$

Последний шаг вытекает из того факта, что всякий объект является своим собственным значением. Поскольку функция значения  $\mu$  в конечном счете вычисляет все применения, то можно рассматривать  $\text{apply} : \langle \text{NULL}, A \rangle$  как порождение  $F$ , несмотря на то что фактический результат есть  $(\text{NULL} : A)$ .

**13.3.2. Как объекты представляют функции; функция представления  $\rho$ .** Как мы уже видели, некоторые атомы (*примитивные атомы*) будут представлять примитивные функции системы. Другие атомы могут представлять определяемые функции точно так же, как символы делают это в системах ФП. Если атом не является ни примитивным, ни определяемым, он представляет  $\perp$ , функцию, которая повсюду есть  $\perp$ .

Последовательности тоже представляют функции и аналогичны функциональным формам в ФП. Функция, представляемая последовательностью, задается (рекурсивно) следующим правилом.

*Правило метакомпозиции*

$$(\rho \langle x_1, \dots, x_n \rangle) : y = (\rho x_1) : \langle \langle x_1, \dots, x_n \rangle, y \rangle,$$

где  $x$  и  $y$  являются объектами. Здесь  $\rho x_1$  определяет, что представляет функциональная форма  $\langle x_1, \dots, x_n \rangle$ , а  $x_2, \dots, x_n$  являются параметрами формы (в ФФП сам объект  $x_1$  может также служить параметром). Так, например, пусть  $\text{Def } \rho \text{ CONST} \equiv \equiv 2 \circ 1$ ; тогда  $\langle \text{CONST}, x \rangle$  в ФФП представляет функциональную форму  $x$  из ФП, потому что, согласно правилу метакомпозиции, если  $y \neq \perp$ , то

$$\begin{aligned} (\rho \langle \text{CONST}, x \rangle) : y &= (\rho \text{ CONST}) : \langle \langle \text{CONST}, x \rangle, y \rangle \\ &= 2 \circ 1 \langle \langle \text{CONST}, x \rangle, y \rangle = x. \end{aligned}$$

Здесь мы можем видеть, что первый управляющий оператор из последовательности или формы, в данном случае  $\text{CONST}$ , всегда имеет своим операндом после метакомпозиции пару, в которой первый элемент сам является последовательностью, а второй элемент является исходным операндом последовательности, в данном случае  $y$ . Управляющий оператор может переупорядочивать и применять заново элементы последовательности и исходный операнд самыми разнообразными способами. Существенный аспект метакомпозиции состоит в том, что она позволяет описывать новые функциональные формы просто путем определения новых функций. Она позволяет также записывать рекурсивные функции без определения.

Рассмотрим еще один пример контролирующей функции для функциональной формы  $\text{Def } \rho \text{ CONS} \equiv \alpha \text{ apply} \cdot \text{tl} \cdot \text{distr}$ . Это определение приводит к выражению  $\langle \text{CONS}, f_1, \dots, f_n \rangle$  (где  $f_i$  — суть объекты), представляющему ту же функцию, что и  $[\rho f_1, \dots, \rho f_n]$ . Ниже доказывается это утверждение.

$$\begin{aligned}
&(\rho\langle CONS, f_1, \dots, f_n \rangle) : x \\
&= (\rho CONS) : \langle \langle CONS, \dots, f_n \rangle, x \rangle \text{ в силу метакомпозиции} \\
&= \alpha \text{ apply} \cdot \text{tl} \cdot \text{distr} : \langle \langle CONS, f_1, \dots, f_n \rangle, x \rangle \text{ по определению } \rho CONS \\
&= \alpha \text{ apply} : \langle \langle f_1, x \rangle, \dots, \langle f_n, x \rangle \rangle \text{ по определению tl, distr и } \alpha \\
&= \langle \text{apply} : \langle f_1, x \rangle, \dots, \text{apply} : \langle f_n, x \rangle \rangle \text{ по определению } \alpha \\
&= \langle (f_1 : x), \dots, (f_n : x) \rangle \text{ по определению apply.}
\end{aligned}$$

При вычислении последнего выражения семантическая функция  $\mu$  будет порождать значение каждого применения, давая  $\rho f_i : x$  в качестве  $i$ -го элемента.

Обычно при описании функции, представляемой последовательностью, мы будем задавать ее общий эффект, а не показывать, как ее управляющий оператор обеспечивает этот эффект. Таким образом, мы будем просто писать

$$(\rho\langle CONS, f_1, \dots, f_n \rangle) : x = \langle (f_1 : x), \dots, (f_n : x) \rangle$$

вместо более детальной записи, приведенной выше.

Нам нужен управляющий оператор  $COMP$  для получения последовательностей, представляющих композицию функциональных форм. Будем считать, что  $\rho COMP$  является примитивной функцией, такой, что для всех объектов  $x$

$$(\rho\langle COMP, f_1, \dots, f_n \rangle) : x = (f_1 : (f_2 : (\dots (f_n : x) \dots))) \text{ при } n \geq 1.$$

(Я признателен П. Макджонсу за его наблюдение, что обычную композицию можно получить с помощью этой примитивной функции вместо использования двух правил композиции в основной семантике, как было сделано в ранней работе [2].)

Хотя системы ФФП позволяют описывать и исследовать новые функциональные формы, следует ожидать, что большинство программистов ограничились бы использованием фиксированного набора форм (управляющие операторы которых являются примитивными), как в системах ФП, так что можно было бы применять алгебраические законы для этих форм и воспользоваться стилем структурного программирования на основе этих форм.

Помимо использования в определениях функциональных форм, метакомпозиция может служить для непосредственного создания рекурсивных функций без применения определения вида  $\text{Def } f \equiv E(f)$ . Например, если  $\rho MLAST = \text{null} \cdot \text{tl} \cdot 2 \rightarrow 1 \cdot 2$ ;  $\text{apply} \cdot [1, \text{tl} \cdot 2]$ , то  $\rho\langle MLAST \rangle \equiv \text{last}$ , где  $\text{last} : x \equiv x = \langle x_1, \dots, \dots, x_n \rangle \rightarrow x_n$ ;  $\perp$ . Таким образом, оператор  $\langle MLAST \rangle$  работает следующим образом:

$$\begin{aligned}
&\mu(\langle MLAST \rangle : \langle A, B \rangle) \\
&= \mu(\rho MLAST : \langle \langle MLAST \rangle, \langle A, B \rangle \rangle) \text{ в силу метакомпозиции}
\end{aligned}$$

$$\begin{aligned}
&= \mu(\text{apply} \circ [1, \text{tl} \circ 2] : \langle \langle \text{MLAST} \rangle, \langle A, B \rangle \rangle) \\
&= \mu(\text{apply} : \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle) \\
&= \mu(\langle \text{MLAST} \rangle : \langle B \rangle) \\
&= \mu(\rho \text{MLAST} : \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle) \\
&= \mu(1 \circ 2 \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle) \\
&= B.
\end{aligned}$$

**13.3.3. Резюме свойств функций  $\rho$  и  $\mu$ .** Ранее мы показали, как функция  $\rho$  отображает атомы и последовательности на функции и как эти функции отображают объекты на выражения. Фактически функция  $\rho$  и все функции ФФП могут быть обобщены таким образом, что становятся определенными для всех выражений. При таких обобщениях свойства функций  $\rho$  и  $\mu$  могут быть подытожены следующим образом:

- (1)  $\mu \in [\text{выражение} \rightarrow \text{объекты}]$ .
- (2) Если  $x$  — это объект, то  $\mu x = x$ .
- (3) Если  $e$  является выражением и  $e = \langle e_1, \dots, e_n \rangle$ , то  $\mu e = \langle \mu e_1, \dots, \mu e_n \rangle$ .
- (4)  $\rho e \in [\text{выражения} \rightarrow \text{выражения} \rightarrow \text{выражения}]$
- (5) Для любого выражения  $e$ ,  $\rho e = \rho(\mu e)$ .
- (6) Если  $x$  — объект, а  $e$  — выражение, то  $\rho x : e = \rho x : (\mu e)$ .
- (7) Если  $x$  и  $y$  — объекты, то  $\mu(x : y) = \mu(\rho x : y)$ . В словесном выражении это означает, что значение применения  $(x : y)$  в системе ФФП отыскивается применением к  $y$  функции  $\rho x$ , представимой с помощью  $x$ , а затем нахождением значения выражения-результата (которое *обычно* является объектом и в таком случае представляет собой собственное значение).

**13.3.4. Ячейки, доставка и запоминание.** В силу ряда причин удобно строить функции, служащие в роли имен. В частности, нам понадобится эта возможность для описания семантики определений в системах ФФП. Чтобы ввести именующие функции, т. е. возможность *доставить* (*fetch*) содержимое ячейки с данным именем из памяти (последовательности ячеек) и для *запоминания* (*store*) ячейки с данным именем и содержимым в такой последовательности, мы вводим объекты, называемые *ячейками* (*cell*), и две новые функциональные формы *fetch* и *store*.

**Ячейки.** Ячейка представляет собой триплет (*CELL*, *имя*, *содержимое*). Мы используем эту форму вместо пары (*имя*, *содержимое*), так что ячейки можно отличить от обычных пар.

**Доставка.** Функциональная форма *fetch* использует в качестве своего параметра объект  $n$  (обычно  $n$  — это атом, служащий в роли имени); это записывается  $\uparrow n$  (читается «доставить  $n$ »). Описание этой функциональной формы для объектов  $n$  и  $x$  имеет вид

$\uparrow n : x \equiv x = \emptyset \rightarrow \#$ ;  $\text{atom} : x \rightarrow \perp$ ;  $(1 : x) = \langle \text{CELL}, n, c \rangle \rightarrow c$ ;  $\uparrow n \cdot \text{tl} : x$

где  $\#$  — это атом «отсутствие». Таким образом,  $\uparrow n$  (доставить  $n$ ) в применении к последовательности дает содержимое первой ячейки из последовательности, имя которой есть  $n$ ; если не существует ячейки с именем  $n$ , то результатом является отсутствие,  $\#$ . Итак,  $\uparrow n$  — это имя функции для имени  $n$ . (Мы предполагаем, что  $\text{pFETCH}$  — это примитивная функция, такая, что  $\text{p}\langle \text{FETCH}, n \rangle \equiv \uparrow n$ . Заметим, что  $\uparrow n$  просто пропускает элементы в своем операнде, которые не являются ячейками.)

**Запоминание, помещение на стек, снятие со стека, чистка.** Подобно доставке, функциональная форма *store* берет в качестве своего параметра объект  $n$ ; она записывается  $\downarrow n$  («запомнить  $n$ »). При применении к паре  $(x, y)$ , где  $y$  — это последовательность,  $\downarrow n$  удаляет из  $y$  первую ячейку с именем  $n$ , если такая есть, а затем создает новую ячейку с именем  $n$  и содержимым  $x$  и присоединяет её к  $y$ . Прежде чем определять  $\downarrow n$  (запомнить  $n$ ), мы специфицируем четыре вспомогательные функциональные формы. (Они могут быть использованы в сочетании с  $\text{fetch } n$  и  $\text{store } n$  для получения множественных именованных стеков LIFO (последним пришел — первым ушел) в рамках последовательности памяти.) Две из этих вспомогательных форм специфицируются рекурсивными функциональными уравнениями; в каждой из них в роли параметра выступает объект  $n$ . Ниже  $\text{cellname}$  — это имя ячейки,  $\text{push}$  — помещение  $n$  на стек,  $\text{pop } n$  — снятие со стека в  $n$ ,  $\text{purge } n$  — чистка  $n$ .

$$(\text{cellname } n) \equiv \text{atom} \rightarrow \overline{F}; \text{eq} \cdot [\text{length}, \overline{3}] \rightarrow \text{eq} \cdot [\overline{[\text{CELL}, n]}, [1, 2]]; \overline{F}$$

$$(\text{push } n) \equiv \text{pair} \rightarrow \text{apndl} \cdot [\overline{[\text{CELL}, n]}, 1], 2]; \overline{1}$$

$$(\text{pop } n) \equiv \text{null} \rightarrow \overline{\emptyset}; (\text{cellname } n) \cdot 1 \rightarrow \text{tl}; \text{apndl} \cdot [1, (\text{pop } n) \cdot \text{tl}]$$

$$(\text{purge } n) \equiv \text{null} \rightarrow \overline{\emptyset}; (\text{cellname } n) \cdot \rightarrow (\text{purge } n) \cdot \text{tl};$$

$$\text{apndl} \cdot [1, (\text{purge } n) \cdot \text{tl}]$$

$$\downarrow n \equiv \text{pair} (\text{push } n) \cdot [1, (\text{pop } n) \cdot 2]; \overline{1}$$

Приведенные выше функциональные формы работают следующим образом. Для  $x \neq \perp$  имеем  $(\text{cellname } n) : x$  равно  $T$ , если  $x$  — это ячейка с именем  $n$ , или равно  $F$  в противном случае;  $(\text{pop } n) : y$  удаляет первую ячейку с именем  $n$  из последовательности  $y$ ;  $(\text{purge } n) : y$  удаляет из  $y$  все ячейки с именем  $n$ ;  $(\text{push } n) : \langle x, y \rangle$  помещает ячейки с именем  $n$  и содержимым  $x$  в начало последовательности  $y$ ;  $\downarrow n : \langle x, y \rangle$  равно  $(\text{push } n) : \langle x, (\text{pop } n) : y \rangle$ .

(Таким образом,  $(\text{push } n) : \langle x, y \rangle$  помещает  $x$  на вершину

«стека» с именем  $n$  в  $y'$ ;  $x$  можно прочесть с помощью  $\uparrow n : y' = x$  и можно удалить с помощью  $(\text{pop } n) : y'$ . Итак,  $\uparrow n \circ (\text{pop } n) : y' —$  это элемент под  $x$  в стеке  $n$  при условии, что в  $y'$  имеется более чем одна ячейка с именем  $n$ .)

**13.3.5. Определения в системах ФФП.** Семантика в системе ФФП зависит от фиксированного множества определений  $D$  (последовательности ячеек) точно так же, как система ФП зависит от ее неформально заданного множества определений. Таким образом, семантическая функция  $\mu$  зависит от  $D$ ; изменение  $D$  дает новую функцию  $\mu'$ , которая отражает измененные определения. Мы представляли  $D$  как объект, потому что в системах ФСПС (разд. 14) мы захотим преобразовывать множество  $D$  путем применения к нему функций и доставлять из него данные, кроме его использования в качестве источника определений функций в семантике ФФП.

Если  $\langle \text{CELL}, n, c \rangle$  — это первая ячейка с именем  $n$  в последовательности  $D$  (и  $n$  — это атом), то она действует как описание  $\text{Def } n = rc$  в ФП, т. е. значение  $(n : x)$  такое же, как значение  $rc : x$ . Например, если  $\langle \text{CELL}, \text{CONST}, \langle \text{COMP}, 2, 1 \rangle \rangle$  — это первая ячейка в  $D$  с именем  $\text{CONST}$ , то она действует так же, как  $\text{Def } \text{CONST} \equiv 2 \circ 1$ , и при этом множестве  $D$  система ФФП нашла бы  $\mu(\text{CONST} : \langle \langle x, y \rangle, z \rangle) = y$ , и следовательно,  $\mu(\langle \text{CONST}, A \rangle : B) = A$ .

В общем случае в системе ФФП с описаниями  $D$  значение применения формы  $(\text{atom} : x)$  зависит от  $D$ ; если  $\uparrow \text{atom} : D \neq \#$  (т. е.  $\text{atom}$  описан в  $D$ ), то его значение есть  $\mu(c : x)$ , где  $c = \uparrow \text{atom} : D$ , т. е. содержимое первой ячейки в  $D$  с именем  $\text{atom}$ . Если  $\uparrow \text{atom} : D = \#$ , то  $\text{atom}$  не описан в  $D$ , и либо  $\text{atom}$  является примитивом, т. е. система знает, как вычислять  $\rho \text{atom} : x$  и  $\mu(\text{atom} : x) = \mu(\rho \text{atom} : x)$ , либо в противном случае  $\mu(\text{atom} : x) = \perp$ .

#### 13.4. ФОРМАЛЬНЫЕ СЕМАНТИКИ ДЛЯ СИСТЕМ ФФП

Мы предполагаем, что множество  $A$  атомов, множество  $D$  определений, множество  $P \subset A$  примитивных атомов и представляемых ими примитивных функций уже выбраны. Предполагаем также, что  $\rho a$  — это примитивная функция, представляемая  $a$ , если  $a$  принадлежит множеству  $P$ , и что  $\rho a = \perp$ , если  $a$  принадлежит  $Q$ , множеству атомов в  $A - P$ , которые не описаны в  $D$ . Хотя функция  $\rho$  определена для всех выражений (см. 13.3.3), формальная семантика использует ее определение только в множествах  $P$  и  $Q$ . Те функции, которые  $\rho$  присваивает другим выражениям  $x$ , неявно определены и применяются для вычисления  $\mu(x : y)$  по следующим семантическим правилам:



Упомянутый выше выбор  $A$ ,  $D$  и  $P$  и соответствующих примитивных функций определяет объекты, выражения и семантическую функцию  $\mu_D$  для системы ФФП. (Мы считаем множество  $D$  фиксированным и пишем  $\mu$  вместо  $\mu_D$ .) Предполагаем, что  $D$  — это последовательность и что функция  $\uparrow y : D$  может быть вычислена (функция  $\uparrow$  задана в разд. 13.3.4) для любого атома  $y$ . При этих предположениях мы описываем  $\mu$  как наименьшую фиксированную точку функционала  $\tau_\mu$ , где функция  $\tau_\mu$  для любой функции  $\mu$  описывается следующим образом (для всяких выражений  $x$ ,  $x_i$ ,  $y$ ,  $y_i$ ,  $z$  и  $w$ ):

$$\begin{aligned} (\tau_\mu)x &\equiv x \in A \rightarrow x; \\ x &= \langle x_1, \dots, x_n \rangle \rightarrow \langle \mu x_1, \dots, \mu x_n \rangle; \\ x &= (y : z) \rightarrow \\ &\quad (y \in A \ \& \ (\uparrow y : D) = \# \rightarrow \mu((\rho y)(\mu z))); \\ &\quad y \in A \ \& \ (\uparrow y : D) = w \rightarrow \mu(w : z); \\ y &= \langle y_1, \dots, y_n \rangle \rightarrow \mu(y_1 : \langle y, z \rangle); \mu(\mu y : z); \perp \end{aligned}$$

Приведенное выше описание  $\mu$  разлагает оператор применения посредством определений и метакомпозиции, прежде чем вычислить операнд. Предполагается, что в указанном описании  $\tau_\mu$  предикаты виды " $x \in A$ " сохраняют  $\perp$  (например, " $\perp \in A$ " имеет значение  $\perp$ ) и что само условное выражение тоже сохраняет  $\perp$ . Таким образом,  $(\tau_\mu)\perp \equiv \perp$  и  $(\tau_\mu)(\perp : z) \equiv \perp$ . Этим завершается рассмотрение семантики систем ФФП.

## 14. ФУНКЦИОНАЛЬНЫЕ СИСТЕМЫ ПЕРЕХОДОВ СОСТОЯНИЙ (СИСТЕМЫ ФСПС)

### 14.1. ВВЕДЕНИЕ

В этом разделе дается обзор класса систем, упомянутых ранее как альтернативы для систем фон Неймана. Необходимо снова подчеркнуть, что эти функциональные системы перехода состояний выдвигаются на передний план не как практические системы программирования в их современной форме, а как примеры класса, в котором функциональный стиль программирования становится доступным в исторически чувствительных системах, не являющихся системами фон Неймана. Эти системы слабо связаны с состояниями и зависят от лежащей в основе функциональной системы и в отношении их языка программирования, и в отношении описания их переходов состояний. Основная функциональная система описываемой ниже системы ФСПС представляет собой систему ФФП, но могли бы использоваться и другие функциональные системы.

Для понимания причин, мотивирующих выбор структуры систем ФСПС, полезно сначала вспомнить основную структуру одной из систем фон Неймана, а именно Алгола, рассмотреть ее ограничения и сравнить ее со структурой систем ФСПС. После такого обзора описывается минимальная система ФСПС. Представлена небольшая, организованная по принципу сверху вниз, с самозащитой, системная программа для поддержания файлов и организации прохождения пользовательских программ, указываются пути ее реализации в системе ФСПС и выполнения примера программы пользователя. В системной программе используются «именные функции» вместо обычных имен, аналогично может поступать и пользователь. Раздел заключается подразделами, в которых обсуждаются варианты системы ФСПС, их общие свойства и системы именования.

#### 14.2. СРАВНЕНИЕ СТРУКТУРЫ АЛГОЛА СО СТРУКТУРОЙ СИСТЕМ ФСПС

Программа на Алголе является последовательностью операторов, каждый из которых представляет преобразование состояния Алгола, т. е. сложного сочетания информации о состояниях различных стеков, указателей, переменных, отображающих идентификаторы на значения, и т. д. Всякий оператор связывается с этим постоянно изменяющимся состоянием посредством изолированных протоколов, каждый из которых зависит от самого оператора и даже от его составных частей (например, протокол, ассоциирующийся с переменной  $x$ , зависит от ее появления в левой или правой части присваивания, в описании, в виде параметра и т. д.).

Дело обстоит так, будто состояние Алгола представляет собой сложную «память», которая связывается с программой на Алголе через огромный «кабель», состоящий из многих специализированных проводов. Сложные протоколы связей этого кабеля фиксированы и включают в себя протоколы для каждого типа операторов. «Значение» программы на Алголе должно выражаться через общий эффект огромного количества актов коммуникаций с состоянием посредством кабеля и его протоколов (а также через средства идентификации выходных данных и включения входных данных в состояние). По сравнению с этим массивным кабелем, связывающим состояние и память Алгола с программой на Алголе, тот кабель, который представляет «бутылочное горлышко» компьютера фон Неймана, является простым, изысканным понятием.

Таким образом, операторы Алгола не являются выражениями, представляющими функции перехода из одного состояния в другое и строящимися из более простых функций переходов

состояний с использованием упорядоченных комбинационных форм. Напротив, это сложные *сообщения* с зависящими от контекста частями, которые «обгрызаются» состоянием. Каждая часть передает информацию в состояние и из состояния через кабель по его собственным протоколам. Не предусматривается средств для применения общих функций ко *всему* состоянию и тем самым для внесения в него больших изменений. Возможность больших, мощных преобразований состояния  $S$  применением функций  $S \rightarrow f: S$  на самом деле невообразима в контексте кабеля и протоколов фон Неймана: ведь не было бы уверенности, что новое состояние  $f: S$  окажется согласованным с кабелем и его фиксированными протоколами, если только  $f$  не ограничивается мелкими изменениями, которые прежде всего должны допускаться кабелем.

Нам желательна вычислительная система, семантика которой не зависит от множества причудливых протоколов для связи с состоянием, и мы хотим уметь вносить в состояния большие преобразования путем применения общих функций. Системы ФСПС обеспечивают один из способов достижения этих целей. В их семантике имеются два протокола для получения информации из состояния: (1) получение из него определения функций, которую нужно применить, и (2) получение самого состояния в целом. Имеется один протокол для изменения состояния: вычисление нового состояния применением функции. За исключением актов коммуникации с состоянием, семантика ФСПС является функциональной (т.е. соответствует системе ФФП). Она не зависит от изменений состояний, потому что в течение вычисления состояние совсем не изменяется. Наоборот, результат вычислений образует входные данные и новое состояние. Структура состояния системы ФСПС несколько ограничена одним из ее протоколов: должна иметься возможность идентифицировать определение (т.е. ячейку) в этом состоянии. Его структура имеет вид последовательности и гораздо более проста, чем структура состояния в Алголе.

Итак, структура систем ФСПС свободна от сложности и ограничений состояния фон Неймана (с его протоколами связи) и достигает большей мощи и свободы совсем другим и более простым способом.

### 14.3. СТРУКТУРА СИСТЕМЫ ФСПС

Система ФСПС состоит из трех элементов:

- (1) *Функциональная подсистема* (такая, как система ФФП).
- (2) *Состояние D*, представляющее собой множество определений функциональной подсистемы.
- (3) Множество *правил перехода*, описывающих, как вход-

ные данные преобразуются в выходные и как изменяется состояние D.

Язык программирования системы ФСПС совпадает с языком ее функциональной подсистемы. (Всюду далее мы будем предполагать, что последняя является системой ФФП.) Итак, системы ФСПС могут использовать стиль программирования ФП, который мы уже обсуждали. Функциональная подсистема не может изменять состояние D, и оно не изменяется в ходе вычисления выражения. Новое состояние вычисляется одновременно с выходными данными и заменяет собой старое состояние в момент выдачи выходных данных. (Напоминаю, что множество определений D является последовательностью ячеек; имя ячейки — это имя определяемой функции, а ее содержимое — это определяющее выражение. Впрочем, некоторые ячейки могут именовать данные, а не функции; имя  $n$  данных будет употребляться в выражении  $\uparrow n$  (доставка  $n$ ), тогда как имя функции будет использоваться в качестве самого оператора.)

Ниже мы приводим правила переходов для элементарной системы ФСПС, которую мы используем для примеров программ. Пожалуй, это простейшие из многих возможных правил переходов, которые могли бы определять поведение широкого разнообразия систем ФСПС.

### 14.3.1. Правила переходов для элементарной системы ФСПС.

Когда система принимает входные данные  $x$ , она формирует применение ( $SYSTEM : x$ ), а затем занимается получением его значения в подсистеме ФФП с использованием текущего состояния D в качестве множества определений.  $SYSTEM$  — это особое имя некоторой функции, определенной в D (т. е. это «системная программа»). Обычно результатом является пара

$$\mu(SYSTEM : x) = \langle o, d \rangle$$

где  $o$  — это выходные данные системы, которые являются результатом обработки входных данных, а  $d$  становится новым состоянием D для следующих входных данных системы. Обычно состояние  $d$  будет копией или частично измененной копией старого состояния. Если  $\mu(SYSTEM : x)$  не является парой, то на выходе получается сообщение об ошибке и состояние остается неизменным.

**14.3.2. Правила перехода: исключительные условия и начальные действия.** Коль скоро входные данные были получены, наша система не будет принимать других входных данных (кроме  $\langle RESET, x \rangle$ , см. ниже), пока не будут выданы входные данные и не будет установлено новое состояние, если оно возникнет. Система примет входные данные  $\langle RESET, x \rangle$  в любой

момент. Существует два варианта: (а) если *SYSTEM* определена в текущем состоянии *D*, то система прерывает свои текущие вычисления, не изменяя *D*, и интерпретирует *x* как новые обычные входные данные; (б) если *SYSTEM* не определена в *D*, то *x* присоединяется к *D* в качестве первого элемента. (На этом завершается полное описание правил переходов для нашей элементарной системы ФСПС.)

Если *SYSTEM* определена в *D*, то она всегда может воспрепятствовать любому изменению в своем собственном определении. В противном случае обычные входные данные *x* породят  $\mu(\text{SYSTEM}: x) = \perp$  и правила перехода приведут к сообщению об ошибке без изменения состояния. С другой стороны, входные данные  $\langle \text{RESET}, \langle \text{CELL}, \text{SYSTEM}, s \rangle \rangle$  определяют *SYSTEM* как *s*.

**14.3.3. Программный доступ к состоянию; функция  $\rho\text{DEFS}$ .** Требуется, чтобы наша подсистема ФФП обладала одной новой примитивной функцией определений, именуемой *DEFS* и такой, что для любого объекта  $x \neq \perp$

$\text{defs} : x = \rho\text{DEFS} : x = D$

где *D* — текущее состояние и множество определения системы ФСПС. Эта функция представляет программам доступ ко всему состоянию для любой цели, в том числе для особо важной цели вычисления следующего состояния.

#### 14.4. ПРИМЕР СИСТЕМНОЙ ПРОГРАММЫ

Данное выше описание нашей элементарной системы ФСПС, а также подсистемы ФФП, примитивов ФП и функциональных форм из предыдущих разделов специфицирует полную исторически чувствительную вычислительную систему. Ее поведение на входе и выходе ограничивается простыми правилами перехода, но в остальных отношениях это мощная система, коль скоро она оснащена подходящим множеством определений. В качестве примера ее использования мы опишем небольшую системную программу, ее реализацию и работу.

В нашем примере системной программы мы будем оперировать запросами и обновлениями для поддерживаемого ею файла, вычислять выражения ФФП, организовывать выполнение программ любого пользователя, не угрожающих сохранности файла или состояния, и разрешать некоторым избранным пользователям изменять множество описаний и саму системную программу. Все допустимые для нее входные данные будут иметь форму  $\langle \text{key}, \text{input} \rangle$ , где *key* (ключ) — это код, который определяет как класс входных данных (*изменение в системе, выражение, программа, запрос, обновление*), так и личность

пользователя и его допуск к использованию системы применительно к заданному классу входных данных. Мы не станем специфицировать формат для ключа. *Вход* представляет собой соответственно входные данные того класса, который задается *ключом*.

**14.4.1. Общий план системной программы.** Состояние *D* нашей системы ФСПС будет содержать описания всех непримитивных функций, нужных для системной программы и для программ пользователей. (Каждое описание находится в некоей ячейке из последовательности *D*.) Кроме того, имеется ячейка из *D* с именем *FILE*, содержащая *файл*, который поддерживается системой. Мы дадим описания ФП для функций, а потом покажем, как передавать их в систему в форме ФФП. Правила перехода делают входные данные операндом для *SYSTEM*, но мы планируем использовать именные функции для обращения к данным, так что первым делом мы создадим для входных данных две ячейки с именами *KEY* и *INPUT*, содержащие *ключ* и *вход*, и присоединим их к *D*. Эта последовательность ячеек содержит по одной ячейке для ключа, входа и файла; она будет операндом нашей главной функции, называемой подсистемой. Подсистема может тогда получить *ключ*, применив к своему операнду операцию  $\uparrow KEY$  и т. д. Итак, определение

**Def** система  $\equiv \text{pair} \rightarrow \text{подсистема} \circ f; [\text{NONPAIR}, \text{defs}]$

где

$f \equiv \downarrow INPUT \circ [2, \downarrow KEY \circ [1, \text{defs}]]$

побуждает систему выдать *NONPAIR* и сохранить состояние неизменным, если входные данные не являются парой. В противном случае, если это  $\langle key, input \rangle$ , то

$f : \langle key, input \rangle = \langle \langle CELL, INPUT, input \rangle, \langle CELL, KEY, key \rangle, d_1, \dots, d_n \rangle$

где  $D = \langle d_1, \dots, d_n \rangle$ . (Мы могли бы сконструировать иной операнд ровно с тремя ячейками для *ключа*, *входа* и *файла*. Мы не поступили таким образом, потому что реальные программы в отличие от подсистемы содержат много именных функций, относящихся к данным, которые содержатся в состоянии, и эта «стандартная» конструкция операнда пригодна для таких случаев.)

**14.4.2. Функция «подсистема».** Теперь мы дадим описание ФП для функции подсистемы, а затем кратко поясним шесть ее вариантов и вспомогательные функции.

**Def** подсистема  $\equiv$

является-изменением-системы  $\circ \uparrow KEY \rightarrow$  [изменение-отчета, применить] $\circ [\uparrow INPUT, defs]$ ;

является-выражением  $\circ \uparrow KEY \rightarrow [\uparrow INPUT, defs]$ ;

является-программой  $\circ \uparrow KEY \rightarrow$  проверка-системы  $\circ$  применить  $\circ (\uparrow INPUT, defs]$ ;

является-запросом  $\circ \uparrow KEY \rightarrow$  [ответ-на-запрос  $\circ [\uparrow INPUT, \uparrow FILE]$ , defs];

является-обновлением  $\circ \uparrow KEY \rightarrow$

сообщение-об-обновлении,  $\downarrow FILE \circ$  [обновить, defs] $\circ$   
 $[\uparrow INPUT, \uparrow FILE]$ ;

[сообщение-об-ошибке  $\circ [\uparrow KEY, \uparrow INPUT]$ , defs].

В этой подсистеме имеются пять фраз " $p \rightarrow f$ " и заключительная функция по умолчанию для всех шести классов входных данных; интерпретации всех классов приводятся ниже. Напоминаю, что операндом является последовательность ячеек, содержащая *key*, *input*, *file*, а также все определенные функции из D, и что подсистема: *operand* =  $\langle output, newstate \rangle$ .

**Неверные входные данные.** В этом случае результат задается последней функцией из определения, когда ключ не удовлетворяет ни одной из предшествующих фраз. Выходными данными является сообщение об ошибке:  $\langle key, input \rangle$ . Состояние остается неизменным, так как оно задается посредством *defs*: *operand* = D. (Мы предоставляем читателю самому придумать, что именно функция сообщение-об-ошибке должна генерировать из его операнда.)

**Входные данные типа изменение-системы.** Когда имеет место изменение-системы  $\circ \uparrow KEY : operand = \text{изменение-системы} : key = T$ ,

то ключ *key* указывает, что пользователь имеет допуск для внесения изменения в систему, а также то, что *input* =  $\uparrow INPUT : operand$  представляет функцию *f*, которая должна быть применена к D для получения нового состояния  $f : D$ . (Разумеется,  $f : D$  может оказаться бесполезным новым состоянием; на него не налагается никаких ограничений.) Выходом является сообщение, а именно сообщение-об-обновлении  $\langle input, D \rangle$ .

**Входные данные типа выражение.** Когда является-выражением: *key* = T, то система понимает, что результатом должно быть значение ФФП-выражения *input*;  $\uparrow INPUT : operand$  порождает этот результат, и проводится вычисление, как для всяких выражений. Состояние остается неизменным.

**Входные данные типа программы и самозащита системы.** Когда является-программой:  $key = T$ , то и выходные данные, и новое состояние задаются так:  $(p\ input) : D = \langle output, newstate \rangle$ . Если  $newstate$  (новое состояние) содержит файл в надлежащем условии, а также определения системы и другие защищенные функции, то проверка-системы:  $\langle output, newstate \rangle = \langle output, newstate \rangle$ . В противном случае проверка-системы:  $\langle output, newstate \rangle = \langle error-report, D \rangle$ .

Хотя входные данные *program* могут вносить значительные, возможно губительные, изменения в состояние, когда порождается *newstate*, все же в действии проверка-системы можно использовать любой критерий, чтобы позволить ему стать действительно новым состоянием либо сохранить прежнее. Более изощренная проверка-системы могла бы корректировать только запрещенные изменения состояния. Такого рода функции осуществимы, потому что всегда возможен доступ к прежнему состоянию для сравнения с намеряющимся новым состоянием и контроль окончательного разрешения замены состояния.

**Входные данные типа запросов к файлам.** Если является-запросом:  $key = T$ , то разрабатывается функция ответ-на-запрос для получения выходных данных, являющихся ответом на входные данные запроса от его операнда  $\langle input, file \rangle$ .

**Входные данные типа обновления файлов.** Если является-обновлением:  $key = T$ , то вход специфицирует изменение состояния, понимаемое функцией *update* (обновление), которая вычисляет  $updated-file = \text{обновление} : \langle input, file \rangle$ . Таким образом, для  $\downarrow FILE$  операндом является  $\langle updated-file, D \rangle$ , и поэтому  $\downarrow FILE$  запоминает обновленный файл в ячейке *FILE* в новом состоянии. Остальная часть состояния не изменяется. Функция сообщение-об-обновлении генерирует выходные данные на основании своего операнда  $\langle input, file \rangle$ .

**14.4.3. Реализация системной программы.** Мы описали функцию, называемую системой, посредством некоторых определений ФП (пользуясь вспомогательными функциями, поведение которых подробно не описывалось, а только указывалось). Предположим, что имеются определения ФП для всех требующихся непримитивных функций. Тогда можно преобразовать каждое определение, чтобы получить соответствующее имя и содержимое ячейки в *D* (разумеется, само это преобразование выполняется бы другой, лучшей системой). Преобразование завершается заменой всякого имени функции ФП на эквивалентный атом (например, обновление приобретает вид *UPDATE*) и заменой функциональных форм на последовательности, в которых первый член является управляющей функцией для конк-



ретной формы. Так,  $\downarrow FILE \circ [update, defs]$  преобразуется в  $\langle COMP, \langle STORE, FILE \rangle, \langle CONS, UPDATE, DEFS \rangle \rangle$

и функция ФП является такой же, как функция, представляемая объектом ФФП, при условии, что обновление  $\equiv \rho UPDATE$  и  $COMP$ ,  $STORE$  и  $CONS$  представляют управляющие функции для композиции, запоминания и конструкции.

Все определения ФП, нужные для нашей системы, могут быть преобразованы в ячейки, как отмечалось выше; в результате получается последовательность  $D_0$ . Мы предполагаем, что у системы ФСПС имеется пустое состояние, с которого можно начать; поэтому  $SYSTEM$  не определяется. Мы хотим дать начальное определение  $SYSTEM$ , чтобы эта система реализовывала свои следующие данные как состояние; после этого мы можем ввести  $D_0$  и будут реализованы все наши описания, включая саму нашу программу-систему. Для осуществления этого введем первые входные данные:

$\langle RESET, \langle CELL, SYSTEM, loader \rangle \rangle$

где

$loader \equiv \langle CONS, \langle \langle CONST, DONE \rangle, ID \rangle \rangle$ .

Тогда, согласно правилу перехода для  $RESET$ , когда  $SYSTEM$  не определена в  $D$ , ячейка в наших входных данных помещается в начало  $D = \emptyset$ , определяя тем самым  $\rho SYSTEM \equiv \rho loader \equiv \equiv [DONE, id]$ . Наши вторые входные данные — это  $D_0$ , т. е. то множество определений, которое мы хотим сделать состоянием. Обычно правило перехода побуждает систему ФСПС вычислить  $\mu (SYSTEM : D_0) \cdot [DONE, id] : D_0 = \langle DONE, D_0 \rangle$ .

Таким образом, выходным значением для ваших вторых входных данных является  $DONE$ , новое состояние есть  $D_0$ , а  $\rho SYSTEM$  теперь представляет собой нашу системную программу (которая принимает входные данные только вида  $\langle key, input \rangle$ ).

Наша следующая задача состоит в том, чтобы загрузить файл (мы получаем начальное значение  $file$ ). Чтобы загрузить его, мы вводим  $program$  во вновь реализованную систему, которая содержит  $file$  как константу и запоминает эту константу в состоянии; входные данные имеют вид

$\langle program-key, [DONE, store-file] \rangle$

где

$\rho store-file \equiv \downarrow FILE \circ [file, id]$ .

$Program-key$  идентифицирует  $[DONE, store-file]$  как программу, которую нужно применить к состоянию  $D_0$  для получения

выходных данных и нового значения  $D_1$ , которое имеет вид

$pstore\text{-}file : D_0 = \downarrow FILE \circ [\overline{file}, id] : D_0$

или  $D_0$  с ячейкой, содержащей в своем начале *file*. Выходные данные — это  $\overline{DONE} : D_0 = DONE$ . Мы предполагаем, что проверка-системы оставит пару  $\langle DONE, D_1 \rangle$  неизменной. Выше применялись выражения ФП вместо обозначаемых ими объектов ФФП, например  $\overline{DONE}$  вместо  $\langle CONST, DONE \rangle$ .

**14.4.4. Применение системы.** Мы ничего не сказали о том, как структурируются системы, запросы или обновления, и поэтому не можем привести подробный пример операций с файлом. Однако из структуры подсистемы ясно видно, как ответ системы на запросы и обновления зависит от функций ответ-на-запрос, обновление и сообщение-об-обновлении.

Предположим, что матрицы  $m, n$  с именами  $M$  и  $N$  запомнены в  $D$  и что описанная выше функция  $MM$  определена в  $D$ . Тогда входные данные

$\langle expression\text{-}key, (MM \circ [\uparrow M, \downarrow N] \circ DEFS : \#) \rangle$

породят в качестве результата произведение этих двух матриц и неизменное состояние. *Expression-key* идентифицирует применение как выражение, которое нужно вычислить, и поскольку  $defs : \# = D$  и  $[\uparrow M, \downarrow N] : D = \langle m, n \rangle$ , то значением выражения является результат  $MM : \langle m, n \rangle$ , представляющий собой выходные данные.

Наша миниатюрная системная программа не содержит средств передачи управления программе пользователя для обработки многих потоков входных данных, но не представило бы большого труда снабдить ее такими средствами с сохранением мониторинга над программой пользователя с возможностью обратной передачи управления.

## 14.5. ВАРИАНТЫ СИСТЕМ ФСПС

Существенным обобщением предложенных выше систем ФСПС была бы система, обеспечивающая комбинационные формы, т. е. «системные формы» для построения новых систем ФСПС из более простых подсистем ФСПС. Иначе говоря, системная форма использовала бы системы ФСПС как параметры и генерировала бы новую систему ФСПС точно так же, как функциональная форма берет в качестве параметров функции и генерирует новые функции. Эти системные формы были бы сходны по своим свойствам с функциональными формами и стали бы операциями удобной «алгебры систем» во многих отношениях аналогично тому, как функциональные формы представ-

ляют собой «операции» алгебры программ. Однако проблема отыскания удобных системных форм гораздо более трудна, поскольку они должны обрабатывать *RESETS*, сопоставлять входные данные с выходными и комбинировать исторически чувствительные системы, а не фиксированные функции.

Кроме того, полезность или нужность системных форм менее очевидна, чем в случае функциональных форм, когда она имеет существенное значение для построения большого разнообразия функций из исходного множества примитивов, причем даже при отсутствии системных форм средства построения систем ФСПС уже настолько богаты, что можно было бы строить фактически любую систему (при общих свойствах ввода и вывода, предоставляемых заданной схемой ФСПС). Быть может, системные формы оказались бы полезными для построения систем со сложными упорядочениями входных и выходных данных.

#### 14.6. ЗАМЕЧАНИЯ О СИСТЕМАХ ФСПС

Как я пытался показать выше, возможны бесчисленные вариации составных частей системы ФСПС — в том, как она работает, как обрабатывает ввод и вывод, как и когда порождает новые состояния и т. д. В любом случае ряд замечаний уместен для любой осмысленной системы ФСПС:

(а) Изменение состояния происходит однократно для каждого большого вычисления и может обладать полезными математическими свойствами. Изменения состояний не связаны с мелкими подробностями вычисления, как в традиционных языках, так что устраняется лингвистическая «узость фон Неймана». Для связи с состоянием не требуется сложного «кабеля» или протоколов.

(б) Программы пишутся на аппликативном языке, и он может включать большое разнообразие изменяемых частей, мощь и гибкость которых превосходит возможности любого из существующих языков фон Неймана. Стиль «слово за словом» заменяется на функциональный стиль, отсутствует разделение программирования на мир выражений и мир операторов. Программы могут анализироваться и оптимизироваться посредством алгебры программ.

(в) Поскольку состояние не может измениться в ходе вычисления выражения система:  $x$ , то не возникает побочных эффектов. Поэтому независимые применения могут вычисляться параллельно.

(г) Я полагаю, что посредством описания подходящих функций можно вводить в любой момент важные новые свойства, используя ту же структуру. Такие свойства должны реализовыв-

ваться в рамках структуры языка фон Неймана. Я имею в виду такие возможности, как «регистры памяти» с большим разнообразием систем именования, типы и проверки типов, взаимодействующие параллельные процессы, недетерминированность и конструкции «охраняемых команд» Дейкстры [8], а также улучшенные методы структурного программирования.

(д) Структура системы ФСПС включает в себе синтаксис и семантику лежащей в основе функциональной системы в сочетании с намеченной выше системной структурой. По современным стандартам это является очень слабой структурой для языка, и никаких других фиксированных частей система не содержит.

#### 14.7. СИСТЕМЫ ИМЕНОВАНИЯ В МОДЕЛЯХ ФСПС И ФОН НЕЙМАНА

Как отмечалось в разд. 13.3.3, в системе ФСПС именование осуществляется функциями. Можно описать много полезных функций для доступа к памяти и изменения ее содержимого (например, занести на стек, снять со стека, очистить, выбрать и т. д.). Все эти определения и соответствующие им системы именования могут быть введены без изменения структуры ФСПС. В одной и той же программе могут использоваться различные виды «регистров памяти» (например, с ячейками, снабженными типом) со своими системами именования. Ячейка одного регистра может содержать другой регистр целиком.

Важная особенность систем именования ФСПС состоит в том, что они реализуют функциональную природу имен (система GEDANKEN Рейнолдса [19] до некоторой степени обеспечивает то же самое в рамках структуры фон Неймана). Функции-имена могут объединяться и комбинироваться с другими функциями посредством функциональных форм. В языках фон Неймана, наоборот, функции и имена обычно представляют собой не связанные между собой понятия, и сходство природы функций и имен почти полностью игнорируется и не используется, потому что (а) имена не могут применяться как функции; (б) отсутствуют общие средства комбинирования имен с другими именами и функциями; (в) объекты, к которым применяются имена-функции (регистры памяти), недоступны в качестве обычных объектов.

Возможно, одна из основных слабостей языков фон Неймана состоит в их неспособности интерпретировать имена как функции. Во всяком случае, возможность употреблять имена как функции и регистры памяти как объекты может оказаться полезной и важной концепцией программирования, которая заслуживает тщательной разработки.

## 15. ЗАМЕЧАНИЯ О ПРОЕКТИРОВАНИИ КОМПЬЮТЕРОВ

Из-за преобладания языков фон Неймана проектировщики имели в своем распоряжении немного интеллектуальных моделей для практической разработки компьютеров вне рамок вариантов компьютера фон Неймана. Другим классом исторически чувствительных моделей являются модели потоков данных [1, 7, 13]. Правила подстановок языков, основанных на лямбда-исчислении, ставят перед проектировщиком вычислительных машин серьезные проблемы. Берклинг [3] разработал модифицированное лямбда-исчисление с тремя видами применений, в котором нет необходимости переименовывать переменные. Он спроектировал машину для вычисления выражений этого языка. Требуется дополнительная практика, чтобы показать, насколько надежной основой является этот язык для эффективного стиля программирования и сколь эффективной может стать машина Берклинга.

Маго [15] разработал новую функциональную машину, которая строится из идентичных компонентов (двух видов). Она непосредственно вычисляет снизу вверх функциональные выражения типа ФП и других типов. В ней отсутствует «память» фон Неймана и нет регистра адреса, а следовательно, нет и узости; она способна параллельно вычислять много приложений; ее встроенные операции напоминают операторы ФП в большей степени, чем операции компьютера фон Неймана. Из всех известных мне конструкций эта наиболее отделилась от компьютера фон Неймана.

Имеются многочисленные свидетельства того, что функциональный стиль программирования может по своей мощи превзойти стиль фон Неймана. Поэтому для программистов имеет большое значение разработка нового класса исторически чувствительных моделей вычислительных систем, которые воплощают такой стиль и не страдают неэффективностью, по-видимому, присущей системам, основанным на лямбда-исчислении. Только когда такие системы и их функциональные языки докажут свое превосходство над традиционными языками, мы получим экономическую основу для разработки нового вида компьютера, который сможет наилучшим образом реализовать их. Только тогда мы, вероятно, сможем в полной мере использовать большие интегральные схемы в архитектуре компьютеров, не ограниченных узостью фон Неймана.

## 16. ИТОГИ

Пятнадцать предыдущих разделов этой статьи могут быть подытожены следующим образом:

**Раздел 1.** Традиционные языки программирования громоздки, сложны и негибки. Их ограниченные выразительные возможности не оправдывают их пространности и дороговизны.

**Раздел 2.** Модели вычислительных систем, лежащие в основе языков программирования, грубо говоря, распадаются на три класса: (а) простые операционные модели (например, машины Тьюринга), (б) функциональные модели (например, лямбда-исчисление) и (в) модели фон Неймана (например, традиционные компьютеры и языки программирования). Каждому классу моделей присуще свое существенное затруднение: программы класса (а) не структурируемы; модели класса (б) не в состоянии передавать информацию от одной программы к следующей; основания моделей класса (в) не обладают практически полезными свойствами, а программы концептуально бесплодны.

**Раздел 3.** Компьютеры фон Неймана строятся вокруг узкого места — трубки, передающей слово за словом и соединяющей центральное процессорное устройство с памятью. Поскольку программа должна выполнять все свои изменения в памяти посредством перекачивания множества слов туда и обратно через эту «узость фон Неймана», мы воспитаны на стиле программирования, в котором основное внимание уделяется организации потока слов через эту узость, а не более крупным концептуальным единицам наших проблем.

**Раздел 4.** Традиционные языки основываются на стиле программирования для компьютера фон Неймана. Так, переменные = ячейки памяти; операторы присваивания = выборка, запоминание и арифметика; операторы управления = команды передачи управления и проверки условия. Символ «: =» является лингвистической «узостью фон Неймана». Программирование на традиционном (фон Неймана) языке по-прежнему озабочено потоком слов (по одному за раз) через слегка усовершенствованную узость. К тому же языки фон Неймана расщепляют программирование на мир выражений и мир операторов. Первый из них упорядочен, а второй хаотичен и лишь слегка упрощен структурным программированием, которое оставило неза тронутыми основные проблемы самого расщепления и пословного стиля традиционных языков.

**Раздел 5.** В этом разделе сравниваются программа фон Неймана и функциональная программа для вычисления внутренне-го произведения. Иллюстрируется ряд трудностей, присущих первой программе, и ряд преимуществ второй: например, программа фон Неймана является итеративной и пословной, работает только с двумя векторами  $a$  и  $b$  заданной длины  $n$  и мо-

жет стать общей только за счет использования описания процедуры, которое характеризуется сложной семантикой. Функциональная программа является неитеративной, работает с векторами как с объектами, более иерархически сконструирована; она вполне общая и создает операции «внутреннего хозяйства» композицией операторов внутреннего хозяйства более высокого уровня. Она не именуется своими аргументами и поэтому не требует описания процедуры.

**Раздел 6.** Язык программирования включает в себя структуру вместе с некоторыми изменяемыми частями. Структура языка фон Неймана требует, чтобы большинство средств были встроены в нее; она может освоить только ограниченные изменяемые части (например, определяемые пользователем процедуры), потому что для всех нужд изменяемых частей, а также для всех возможностей, встроенных в структуру, должно быть предусмотрено детальное обеспечение в «состоянии» и в его правилах перехода. Структура фон Неймана является столь негибкой именно потому, что ее семантика слишком сильно привязана к состоянию: всякая подробность вычислений изменяет состояние.

**Раздел 7.** Выразительная сила изменяемых частей в языках фон Неймана мала; именно по этой причине большинство средств языка должны быть встроены в структуру. Недостаточность выразительной силы следует из того, что в языках фон Неймана отсутствует возможность эффективно использовать для построения программ комбинационные формы, а это, в свою очередь, следует из расщепления языка на мир выражений и мир операторов. Комбинационные формы лучше всего проявляют себя в выражениях, но в языках фон Неймана выражение способно порождать только одиночное слово; поэтому выразительная сила в мире выражений по большей части теряется. Другим препятствием применению комбинационных форм являются хитроумные соглашения об именовании.

**Раздел 8.** APL — это первый не основанный на лямбда-исчислении язык, который работает не «слово за словом» и использует функциональные комбинационные формы. Однако в нем все еще сохраняются многие из проблем, присущих языкам фон Неймана.

**Раздел 9.** Язык фон Неймана не представляет удобных средств для рассуждений о программах. Аксиоматические и денотационные семантики являются точным инструментарием для описания и понимания традиционных программ, но они лишь позволяют рассуждать о программах и не делают их более удобными. В отличие от языков фон Неймана язык обычной

алгебры удобен как для формулирования ее законов, так и для преобразования уравнений в их решения, и все это в рамках одного «языка».

**Раздел 10.** В исторически чувствительном языке одна программа может влиять на поведение другой, последующей, за счет изменения состояния части памяти, которая сохраняется системой. Любой такой язык требует некоторой семантики переходов состояний. Но ему не нужна семантика, тесно привязанная к состояниям, в которой состояние изменяется с каждой подробностью вычислений. Функциональные системы переходов состояний (ФСФС) предлагаются в качестве исторически чувствительной альтернативы для систем фон Неймана. Они характеризуются: (а) свободно привязанной семантикой переходов состояний, в которой переход происходит лишь однажды для большого вычисления; (б) простыми состояниями и правилами перехода; (в) базовой функциональной системой с простой семантикой «сведения», (г) языком программирования и правилами перехода состояний, основывающимися на базовой функциональной системе и ее семантике. В следующих четырех разделах излагаются элементы этого подхода к проектированию не-фон-неймановских языков и систем.

**Раздел 11.** Описывается класс систем неформального функционального программирования (ФП) без использования переменных. Каждая система строится из объектов, функций, функциональных форм и определений. Функции отображают объекты на объекты. Функциональные формы комбинируют существующие функции для формирования новых. В этом разделе перечисляются примеры примитивных функций и функциональных форм и приводятся примеры программ. Обсуждаются ограничения и преимущества систем ФП.

**Раздел 12.** Описывается «алгебра программ», в которой переменными являются функции систем ФП, а операции — это функциональные формы системы. Список двадцати четырех законов алгебры сопровождается примером доказательства эквивалентности неитеративной программы умножения матриц и рекурсивной программы для той же цели. В следующем подразделе формулируются результаты двух «теорем о разложении», которые «решают» два класса уравнений. Эти решения выражают «неизвестную» функцию из таких уравнений в виде бесконечного условного выражения, которое устанавливает ситуационное описание ее поведения и непосредственно дает необходимые и достаточные условия для завершения. Эти результаты используются для вывода «теоремы рекурсии» и «теоремы итерации», которые обеспечивают готовые к употреблению



выражения для некоторых довольно общих и полезных классов «линейных» уравнений. Примеры применения этих теорем включают: (а) доказательства корректности для рекурсивной и итеративной факториальных функций и (б) доказательства эквивалентности двух итеративных программ. Заключительный пример относится к «квадратному» уравнению и содержит доказательства того, что его решение представляет собой идемпотентную функцию. В следующем подразделе приводятся доказательства двух теорем о разложении.

Ассоциированная с системами ФП алгебра сопоставляется с соответствующими алгебрами для лямбда-исчисления и других функциональных систем. Сопоставление показывает некоторые преимущества несколько ограниченных систем ФП по сравнению с гораздо более мощными классическими системами. Ставятся некоторые вопросы относительно алгоритмического сведения функций бесконечных выражений и относительно использования этой алгебры в различных схемах «ленивого вычисления».

**Раздел 13.** В этом разделе описываются системы формального функционального программирования, которые обобщают и уточняют поведение систем ФП. Их семантики проще семантик классических систем, и их непротиворечивость можно показать простым рассуждением.

**Раздел 14.** В этом разделе сравниваются структуры Алгола и функциональных систем переходов состояний (ФСПС). Описывается система ФСПС, использующая систему ФФП в качестве своей функциональной подсистемы. Для этой системы вводятся простое состояние и правила перехода. Описывается малая самозащищающаяся системная программа для данной системы ФСПС и рассматривается возможность ее реализации и применения для поддержки и организации вычислений по программам пользователей. Кратко обсуждаются варианты систем ФСПС и функциональных систем, которые могут быть описаны и использованы в рамках системы ФСПС.

**Раздел 15.** В этом разделе кратко рассматриваются работа по проектированию функциональных компьютеров и потребность в разработке и тестировании более практичных моделей функциональных систем как будущей основы такого проектирования.

## БЛАГОДАРНОСТИ

В прежней работе, связанной с этой статьей, я получил весьма значительную помощь и много советов от Пола Макджонса и Барри К. Розена. Я также получил очень ценную по-

мощь и советы при подготовке этой статьи. Дж. Н. Грей исключительно щедро делился своими знаниями и потратил много времени на рецензирование первого варианта статьи. Стефен Н. Зиллис тоже тщательно прочитал его. Оба предложили много важных советов и замечаний на этом трудном этапе. Я пользуюсь приятной возможностью выразить им свою признательность. Для меня были полезны также обсуждения первого варианта с Ренальдом Фейджином, Полом Р. Мак-Джонсом и Джеймсом Моррисом. Фейджин предложил ряд усовершенствований доказательства теорем.

Поскольку большая часть статьи содержит технический материал, я попросил двух разных специалистов по информатике прорецензировать третий вариант. Дэвид Грис и Джон Рейнольдс оказались настолько любезны, что взяли на себя этот обременительный труд. Оба предоставили мне большие подробные списки исправлений и общих замечаний, которые привели ко многим большим и малым улучшениям этого последнего варианта (прорецензировать который им не представилось возможности). Я искренне благодарен за затраченные ими время и усилия.

Наконец, я послал копии третьего варианта Джуле Маго, Питеру Науру и Джону Уильямсу. Они любезно откликнулись, прислав ряд исключительно ценных замечаний и исправлений. Джеффри Франк и Дэйв Тоули из Университета Северной Каролины ознакомились с копией, посланной мною Маго, и указали существенную ошибку в описании семантических функций систем ФФП. Всем им я глубоко признателен за оказанную мне помощь.

#### ЛИТЕРАТУРА

1. Arvind and Gostelow K. P. A new interpreter for data flow schemas and its implications for computer architecture. Tech. Rep. No. 72, Dept. Comptr. Sci., U. of California, Irvine, Oct. 1975.
2. Backus J. Programming language semantics and closed applicative languages. Conf. Record ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, 71—86.
3. Berkling K. J. Reduction languages for reduction machines. Interner Bericht ISF—76—8, Gesellschaft für Mathematik und Datenverarbeitung MBH, Sept. 1976.
4. Burge W. H. Recursive Programming Techniques. Addison-Wesley, Reading Mass., 1975.
5. Church A. The Calculi of Lambda-Conversion. Princeton U. Press, Princeton, N. J., 1941.
6. Curry H. B. and Feys R. Combinatory Logic, Vol. I. North-Holland Pub. Co., Amsterdam, 1958.
7. Dennis J. B. First version of a data flow procedure language. Tech. Mem. No. 61, Lab. for Comptr. Sci., M. I. T., Cambridge, Mass., May 1973.
8. Dijkstra E. W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N. J., 1976. [Имеется перевод: Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978. — 274 с.

9. Friedman D. P. and Wise D. S. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, S. Michaelson and R. Milner, Eds., Edinburgh U. Press, Edinburgh, 1976, pp. 257—284.
10. Henderson P. and Morris J. H. Jr. A lazy evaluator. *Conf. Record 3rd ACM Symp. on Principles of Programming Languages*, Atlanta, Ga., Jan. 1976, pp. 95—103.
11. Hoare C. A. R. An axiomatic basis for computer programming. *Comm. ACM* 12, 10 (Oct. 1969), 576—583.
12. Iverson K. A. *Programming Language*. Wiley, New York, 1962.
13. Kosinski P. A data flow programming language. Rep. RS 4264, IBM T. J. Watson Research Ctr., Yorktown Heights, N. Y., March 1973.
14. Landin P. J. The mechanical evaluation of expressions. *Computer J.* 6, 4 (1964), 308—320.
15. Mago G. A. A network of microprocessors to execute reduction languages. To appear in *Int. J. Computr. and Inform. Sci.*
16. Manna Z., Ness S., and Vuillemin J. Inductive methods for proving properties of programs. *Comm. ACM* 16, 8 (Aug. 1973), 491—502.
17. McCarthy J. Recursive functions of symbolic expressions and their computation by machine, Pt. 1. *Comm. ACM* 3, 4 (April 1960), 184—195.
18. McJones P. A. Church—Rosser property of closed applicative languages. Rep. RJ 1589, IBM Res. Lab., San Jose, Calif., May 1975.
19. Reynolds J. C. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM* 13, 5 (May 1970), 308—318.
20. Reynolds J. C. Notes on a lattice-theoretic approach to the theory of computation. *Dept. Syst. and Inform. Sci., Syracuse U., Syracuse, N. Y.*, 1972.
21. Scott D. Outline of a mathematical theory of computation. *Proc. 4th Princeton Conf. on Inform. Sci. and Syst.*, 1970.
22. Scott D. Lattice-theoretic models for various types-free calculi. *Proc. 4th Int. Congress for Logic, Methodology, and the Philosophy of Science*, Bucharest, 1972.
23. Scott D. and Strachey C. Towards a mathematical semantics for computer languages. *Proc. Symp. on Computrs. and Automata*, Polytechnic Inst. of Brooklyn, 1971.

1978

## Парадигмы программирования

*Роберт Флойд*

Станфордский университет

Премия Тьюринга 1978 г. Ассоциации по вычислительной технике (АСМ) была вручена Роберту Флойду председателем Комитета по присуждению премий Уолтером Карлсоном 4 декабря на ежегодной конференции Ассоциации в Вашингтоне.

Осуществляя отбор претендентов, Подкомитет по присуждению премий за крупные научные достижения (бывший Подкомитет по присуждению премии Тьюринга) назвал имя профессора Флойда в связи с его «вкладом в создание следующих важных разделов информатики: теория синтаксического анализа, семантика языков программирования, автоматическая проверка правильности программ, автоматический синтез программ и анализ алгоритмов».

Профессор Флойд, получивший степени бакалавра гуманитарных наук и бакалавра естественных наук в Чикагском университете соответственно в 1953 и 1958 гг., в области информатики является самоучкой. Он начал заниматься этими проблемами в 1956 г., когда, работая ночным оператором на ЭВМ IBM 650, в перерывах между загрузкой карманов для перфокарт смог найти время, чтобы обучиться программированию.

Флойд создал один из первых компиляторов Алгола-60, закончив свою работу над этим проектом в 1962 г. В рамках проекта он выполнил ряд новаторских работ по оптимизации компилятора. Потом вплоть до 1965 г. он занимался систематизацией синтаксического анализа языков программирования. Для этого им были разработаны метод предшествования, метод ограниченного контекста и синтаксический анализ с помощью языка продукций.

В 1966 г. профессор Флойд предложил математический метод доказательства правильности программ. За годы работы он создал ряд полезных быстрых алгоритмов. Среди них — древовидный алгоритм разрядовой сортировки, алгоритмы поиска кратчайших путей в сетях, а также алгоритм нахождения медиан и выпуклых оболочек. Он также определил предельную скорость цифрового суммирования и предельные скорости размещения информации в памяти ЭВМ.

Ему же принадлежит существенный вклад в методы машинного доказательства теорем и создание блоков орфографического контроля.

В последние годы профессор Флойд занимается разработкой и реализацией языка программирования, в первую очередь предназначенного для использования студентами. Язык можно будет использовать при систематическом обучении новичков структурному программированию, и по своим возможностям он будет почти универсален.

\* \* \*

Сегодня я хочу поговорить о парадигмах программирования, об их влиянии на наши успехи как создателей компьютерных программ, о том, как нужно обучать парадигмам и как их следует включать в наши языки программирования.

Известным примером парадигмы программирования служит методика *структурного программирования*, которая, видимо, является доминирующей в большинстве современных подходов. Структурное программирование, как оно определяется Дейкстрой [6], Виртом [27, 29], Парнасом [21] и другими, содержит две фазы.

В первой фазе, представляющей собой нисходящее проектирование или пошаговую детализацию, задача разбивается на весьма незначительное число более простых подзадач. Например, при написании программы решения системы линейных уравнений первым уровнем разбиения станет этап приведения системы к треугольной форме с последующим этапом, заключающимся в обратной подстановке в систему, приведенную к треугольной форме. Это последовательное разбиение продолжается до тех пор, пока возникающие подзадачи не станут достаточно простыми, чтобы их можно было решать непосредственно. В примере с системой линейных уравнений операция обратной подстановки должна подвергаться дальнейшему разбиению в виде обратных итераций некоторого процесса поиска и запоминания значения  $i$ -й переменной из  $i$ -го уравнения. Дальнейшее разбиение должно привести к полностью детализованному алгоритму.

Вторая фаза парадигмы структурного программирования предусматривает движение снизу вверх, от конкретных объектов и функций данной машины к более абстрактным объектам и функциям, используемым в модулях, созданных путем нисходящего проектирования. Если в примере с линейным уравнением коэффициенты уравнений являются рациональными функциями одной переменной, то сначала можно построить арифметическое представление с многократной точностью и соответствующие процедуры, а затем на этой основе — полиномиаль-

ное представление с собственными арифметическими операциями и т. д. Этот подход получил название метода *уровней абстракции* или *сокрытия информации*.

Парадигму структурного программирования никак нельзя считать повсеместно признанной. Наиболее ярые ее сторонники должны признать, что самой по себе ее недостаточно, чтобы превратить все сложные проблемы в простые. Сохраняют свое значение и другие более специализированные парадигмы высокого уровня, такие, как метод ветвей и границ [17, 20] или метод «разделяй и властвуй» [1, 11]. Тем не менее парадигма структурного программирования действительно помогает расширить возможности программиста, позволяя создавать программы, которые слишком сложны, чтобы их можно было писать эффективно и надежно без методологической базы.

Я считаю, что современное состояние дел в программировании отражает неадекватность наших запасов парадигм, нашего знания существующих парадигм, наших методов обучения парадигмам программирования и способа, которым наши языки программирования поддерживают или не поддерживают парадигмы сообществ их пользователей.

Нынешнее состояние искусства программирования было недавно охарактеризовано Робертом Бэлзером [3] следующими словами: «Хорошо известно, что программное обеспечение находится в жалком виде. Оно ненадежно, появляется с опозданием, не реагирует на происшедшие перемены, неэффективно и дорого. Более того, поскольку в настоящее время его создание трудоемко, то ситуация будет и дальше ухудшаться по мере возрастания спроса и увеличения расходов на рабочую силу». Если это напоминает знаменитый «кризис программного обеспечения» примерно десятилетней давности, то тот факт, что на протяжении 10 или 15 лет мы пребываем в одном и том же состоянии, подсказывает, что термин «упадок программного обеспечения» был бы более подходящим.

Томас Кун в книге «Структура научных революций» [16] описывает научные революции за последние несколько столетий как возникавшие из изменений в доминировавших парадигмах. Некоторые из наблюдений Куна выглядят применимыми к нашей области. Относительно учебников, излагающих современные научные знания студентам, Кун пишет:

«Эти тексты, например, часто наводят на мысль, что содержание науки можно единственным образом проиллюстрировать описываемыми на их страницах наблюдениями, законами и теориями».

Аналогично большинство учебников программирования исходит из допущения, что содержание программирования заключается только в знании алгоритмов и определений языков, описываемых на их страницах.

Кун также отмечает:

«Именно изучение парадигм, в том числе парадигм гораздо более специализированных, чем названные мною здесь в целях иллюстрации, служит основным фактором, подготавливающим студента к членству в том или ином научном сообществе. Поскольку в дальнейшем он оказывается в окружении людей, которые изучали основы предмета на тех же самых конкретных моделях, что последующая практика в научных исследованиях не часто будет побуждать его к резкому расхождению с фундаментальными принципами...»

В информатике можно видеть несколько таких сообществ, каждое из которых говорит на собственном языке и использует собственные парадигмы. Фактически каждый из языков программирования обычно поощряет использование одних парадигм и затрудняет использование других. Существуют четко определенные школы программирования на языке Лисп, АPL, Алгол и т. д. В качестве исходной структурной информации о программе одни рассматривают поток данных, а другие — поток управления. Рекурсия и итерация, копирование и совместное использование структур данных, вызов по имени и вызов по значению — все они имеют своих сторонников.

И еще одна цитата из Куна:

«...Старые школы постепенно исчезают. Исчезновение этих школ частично обусловлено обращением их членов к новой парадигме. Но всегда остаются ученые, верные той или иной устаревшей точке зрения. Они просто выпадают из дальнейших совокупных действий представителей их профессии, которые с этого времени игнорируют все их усилия».

В информатике не существует механизма, который исключал бы таких людей из сферы их профессиональной деятельности. Я подозреваю, что они преимущественно становятся менеджерами в сфере разработки программного обеспечения.

Бэлзер в своем плаче о состоянии разработки программного обеспечения пророчествовал, что автоматическое программирование спасет нас. Я желаю успеха сторонникам автоматического программирования, но, пока они не вычистят «конюшен», наша главная надежда связана с расширением собственных возможностей. Наилучшим шансом, которым мы располагаем для улучшения общей практики программирования, я считаю внимательное рассмотрение наших парадигм.

В начале 60-х годов синтаксический анализ бесконтекстных языков представлял собой особенно острую проблему как для разработки компиляторов, так и для лингвистики. Опубликованные алгоритмы были и медленными, и некорректными. Джон Кок (как утверждают, приложив весьма незначительные усилия) нашел быстрый и простой алгоритм [2], основывающийся на теперь уже стандартной парадигме, которая представляет собой вычислительную форму динамического программирования [1]. Парадигма динамического программирования

решает некую задачу для данной входной информации, решая ее вначале итерационно для всех отрезков вводимой информации меньшей длины. Алгоритм Кока успешно осуществлял любые синтаксические анализы всех подцепочек ввода. В этих концептуальных рамках данная проблема становилась почти тривиальной. Полученная процедура стала первым алгоритмом, стабильно завершающим работу при любом входе за полиномиальное время.

Приблизительно тогда же, после публикации нескольких некорректных нисходящих синтаксических анализаторов, этой проблемой занялся и я. Для этого я создал парадигму нахождения иерархической организации процессоров, напоминающей организацию людей-работодателей, нанимающих и увольняющих сотрудников для решения конкретных задач, и затем смоделировал ее работу [8]. Моделирование таких множественных рекурсивных процессов привело меня к использованию в качестве управляющей структуры рекурсивных сопрограмм. Позже я выяснил, что другие программисты, столкнувшись со сложными комбинаторными проблемами, например Гелернтер со своей машиной для доказательства геометрических теорем [10], очевидно, изобрели такую же управляющую схему.

Опыт Джона Кока и мой собственный наглядно показывают вероятность того, что для обеспечения прогресса в области программирования необходимо будет продолжать изобретать, совершенствовать и публиковать новые парадигмы.

Примером эффективной тщательно разработанной парадигмы является работа Шортлифа и Дэвиса над программой MYCIN [24], которая блестяще диагностирует и дает рекомендации по лечению бактериальных инфекций. MYCIN представляет собой систему на основе правил продукции, в которую заложен большой набор независимых правил с проверяемыми условиями применимости каждого правила и простым действием, осуществляемым каждым правилом в случае выполнения соответствующего условия. Программа TERESIAS [5] Дэвиса модифицирует программу MYCIN, позволяя пользователю-эксперту улучшать ее рабочие характеристики. Программа TERESIAS уточняет парадигму, прослеживая ответные действия в обратной последовательности — от ошибочного результата через правила и условия, разрешившие его применение, до того момента, когда будет обнаружено неудовлетворительное правило, дающее неверные результаты из правильных предположений. За счет этого стало технически возможным специалисту-медику, не знакомому с программированием, улучшать диагностические способности программы MYCIN. Хотя в программе MYCIN нет ничего, что не могло бы быть закодировано в виде традиционного ветвящегося дерева решений с использованием условных



переходов, именно использование парадигмы, основанной на системе правил, с последующим ее усовершенствованием добавлением возможности самоизменения сделало возможным интерактивное улучшение программы.

Если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм, то совершенствование искусства отдельного программиста требует, чтобы он расширял свой *репертуар* парадигм. В рамках собственного опыта построения сложных алгоритмов я выработал определенную технику, оказавшуюся наиболее плодотворной для расширения моих возможностей. После того как поставленная задача решена, я повторно решаю ее с самого начала, прослеживая и повторяя только суть предыдущего решения. Я проделяваю эту процедуру до тех пор, пока решение не становится настолько четким и ясным, насколько это для меня возможно. Затем я ищу общее правило решения аналогичных задач, которое *заставило бы* меня подходить к решению поставленной задачи наиболее эффективным способом с первого раза. Часто такое правило приобретает непреходящее значение. Поиски такого общего правила привели меня от ранее упоминавшегося алгоритма синтаксического анализа, основывающегося на рекурсивных программах, к общему методу написания недетерминированных программ [9], которые затем с помощью макрорасширения преобразуются в обычные детерминированные программы. Эта парадигма позднее, оказавшись включенной в такие языки программирования, как PLANNER [12, 13], MICROPLANNER [25] и QA4 [23], нашла применение в казалось бы далекой области — автоматическом решении задач, относящемся к вопросам искусственного интеллекта.

Приобщению отдельного программиста к новым парадигмам может способствовать чтение программ, написанных другими. Но такой подход не очень продуктивен, поскольку материал для чтения заимствуется у программистов, работающих, как правило, в той же фирме и отобранных по их умению использовать те же языки и парадигмы. Подтверждением этому служит то, сколь часто наша промышленность адресует предложения о найме не программистам вообще, а программистам, пишущим на Фортране или Коболе. Правила Фортрана можно выучить за несколько часов, связанные с ними парадигмы требуют значительно большего времени, как для того, чтобы их усвоить, так и для того, чтобы от них отвыкнуть.

Помочь может знакомство с программированием, ведущимся по незнакомым правилам. Работая в этом году во время творческого отпуска в МТИ (MIT), я столкнулся с многочисленными примерами широких возможностей, которые извлекают программисты на Лиспе из единственной структуры дан-

ных, используемой также в качестве универсальной синтаксической структуры для всех встречающихся в программах функций и операций и позволяющей им манипулировать программами как данными. Хотя ранее я был энтузиастом синтаксически богатых языков вроде Алгола, сейчас я ясно и наглядно вижу глубокий смысл тьюринговской лекции М. Минского 1969 г. [19], в которой он утверждал, что однородность структуры Лиспа и рекурсия предоставляют программистам такие возможности, содержание которых вполне компенсирует утраты наглядности визуальной формы программ. Я хотел бы получить какой-то разумный синтез этих подходов.

То, что каждый стремится создать новый язык программирования, остается справедливым для настоящего момента в той же степени, что и для 1956 г., когда я начал заниматься информатикой. «Я предпочитаю писать программы, которые помогают писать программы, а не писать их самому» — выбито на стене одного из зданий Станфордского университета. Оценивая ежегодный «урожай» новых языков программирования, полезно классифицировать их по тому, в какой степени они допускают и поощряют использование эффективных парадигм программирования. Когда мы делаем наши парадигмы четкими, то выясняется, что их очень много. Как показал Корделл Грин, механическая генерация алгоритмов поиска и сортировки, таких, как сортировка слиянием и быстрая сортировка, требует свыше ста правил, большинство из которых, вероятно, это парадигмы, знакомые большей части программистов. Часто наши языки программирования не только не помогают, а и мешают нам в использовании даже парадигм низкого уровня. Приведу несколько примеров.

Допустим, мы моделируем динамику популяций системы хищник — жертва, например волков и кроликов. Мы имеем два уравнения

$$W' = f(W, R),$$

$$R' = g(W, R),$$

которые определяют число волков и кроликов к концу какого-то промежутка времени как функцию их количества в его начале.

Общая ошибка начинающих состоит в том, что они пишут:

```
FOR I := --- DO
  BEGIN
    W := f(W, R);
    R := g(W, R)
  END
```

где  $g$  ошибочно вычисляется с использованием измененного значения  $W$ . Для того чтобы программа заработала, мы должны написать

```
FOR I := --- DO
  BEGIN
    REAL TEMP;
    TEMP := f(W,R);
    R := g(W,R);
    W := TEMP
  END
```

Новичок прав, полагая, что нет необходимости делать это. Одна из наиболее распространенных наших парадигм, как и в случае моделирования системы хищник — жертва, состоит в одновременном присваивании новых значений всем компонентам векторов состояний. Тем не менее едва ли в каком-либо языке существует оператор для одновременного присваивания. Вместо этого нам приходится выполнять чисто техническую, связанную с потерей времени и возможными ошибками операцию заведения одной или нескольких временных переменных и присваивать новые значения через них.

Еще раз обратимся к простой на первый взгляд задаче:

Вводить строки текста до тех пор, пока не обнаружится строка, состоящая только из пробелов. Устранить лишние пробелы между словами. Напечатать текст по тридцать литер в строке, не разрывая слова между строками.

Поскольку ввод и вывод удобно записываются с помощью множественных уровней итерации и поскольку итерации на входе не вкладываются в итерации на выходе и не включают их, то данная задача для программирования на большинстве языков оказывается на удивление сложной [14]. У новичков решение этой задачи займет в 3—4 раза больше времени, чем рассчитывают преподаватели, и завершится либо полным хаосом, либо кустарной структурой управляющей логики программы, в которой используются явные приращения и условное выполнение для моделирования нужных итераций.

Задача естественным образом формулируется с помощью разбиения на три связанные сопрограммы [4] для ввода, преобразования и вывода потока литер. Однако, за исключением языков моделирования, немногие из наших языков программирования содержат структуры управления сопрограммами, позволяющие запрограммировать данную задачу естественным образом.

Когда язык удобен для реализации парадигмы, я буду говорить, что он *поддерживает* данную парадигму. А в случае если на этом языке парадигма реализуема, но сделать это непросто,

я буду говорить, что он слабо поддерживает парадигму. Как показывают два предыдущих примера, большинство наших языков лишь слабо поддерживают одновременное присваивание и абсолютно не поддерживают сопрограмм, хотя требуемые механизмы значительно проще и полезнее, чем, скажем, процедуры рекурсивного вызова по имени, реализованные семнадцать лет назад в семействе языков Алгол.

Даже парадигма структурного программирования в лучшем случае слабо поддерживается многими из наших языков программирования. Чтобы записать программу решения системы уравнений, нужно иметь возможность написать:

```
MAIN__PROGRAM:
  BEGIN
    TRIANGULARIZE;
    BACK__SUBSTITUTE
  END;
BACK__SUBSTITUTE:
  FOR I := N STEP - 1 UNTIL 1 DO
    SOLVE__FOR__VARIABLE(I);
  SOLVE__FOR__VARIABLE(I);
  ---
  ---
  TRIANGULARIZE:
  ---
  ---
```

Процедуры для арифметических операций с многократной точностью

Процедуры для арифметических операций с рациональными функциями

Описания массивов

В большинстве современных языков записать в указанной последовательности главную программу, процедуры и описания данных нельзя. Требуется некоторая предварительная перекомпоновка текста, выполняемая вручную, которая, вообще говоря, вполне поддается автоматизации. Далее, любые переменные, используемые более чем в одной из процедур с многократной точностью, должны быть глобальными для каждой части программы, где могут использоваться арифметические операции с многократной точностью, что позволяет вопреки принципу сокрытия информации осуществлять случайную модификацию. И наконец, детальное разбиение задачи на иерархию процедур обычно приводит к очень неэффективному коду, хотя большинство процедур, поскольку они вызываются только в одном месте главной программы, могут быть реализованы эффективно с помощью макрорасширения.

Парадигма какого-то еще более высокого уровня абстракции, чем парадигма структурного программирования, пред-

ставляет собой иерархию языков, где программы на языке наивысшего уровня оперируют с наиболее абстрактными объектами и транслируются в программы на языке следующего, более низкого уровня. Примерами тому служат многочисленные предназначенные для работы с формулами языки, которые построены на базе Лиспа, Фортрана и других языков. Большинство из наших языков более низкого уровня не могут полностью поддерживать указанные суперструктуры. Например, их системы диагностирования ошибок обычно «отлиты из бетона», так что диагностические сообщения делаются понятными только посредством ссылок транслированный текст программы на языке более низкого уровня.

Я считаю, что дальнейшее развитие программирования как искусства требует разработки и распространения языков, поддерживающих основные парадигмы сообществ их пользователей. Созданию языка должно предшествовать перечисление этих парадигм, включая изучение слабостей программного обеспечения, связанных с неадекватной поддержкой некоторых парадигм. Я не удовлетворен такими расширениями наших языков, как варианты записи и множества подмножеств Паскаля [15, 28], пока парадигмы, о которых я говорил, и многие другие не поддерживаются или слабо поддерживаются. Если когда-нибудь появится наука о проектировании языков программирования, то, вероятно, она в основном будет состоять из сопоставления языков с методами конструирования, которые в них поддерживаются.

Я не хочу утверждать, что поддержка парадигмы ограничивается только возможностями наших языков программирования. Вся среда, в которой мы пишем программы — системы диагностики, системы файлов, редакторы и т. д., — может быть рассмотрена как поддерживающая или не поддерживающая набор методов создания программ. Есть надежда, что это начинает осознаваться. Например, в последней работе в IRIA во Франции и некоторых других используются редакторы, учитывающие структуру программы, которую они редактируют [7, 18, 26]. Оценить это может всякий, кто пытался решить даже столь простую задачу, как изменение всех  $X$ , появляющихся в задаче в качестве идентификатора, без случайного изменения всех других  $X$ .

Теперь я хотел бы поговорить о том, что мы *преподаем* под видом компьютерного программирования. Часть нашей злосчастной одержимости формой в ущерб содержанию, о чем М. Минский сожалел в своей тьюринговской лекции [19], проявляется обычно в выборе того, чему учить. Если спрашиваю какого-либо профессора о том, что он читает в вводном курсе программирования, то ответит ли он с гордостью «Паскаль»

или застенчиво «Фортран», я все равно знаю, что он обучает грамматике, набору семантических правил и некоторым законченными алгоритмам, предоставляя студентам самостоятельно открывать некоторые процессы написания программ. Даже учебники, основанные на парадигме структурного программирования, хотя и содержат «режиссуру» на высшем уровне, который мы можем назвать уровнем «сюжета» в написании программы, часто не оказывают никакой помощи на промежуточных уровнях, которые мы можем назвать уровнем «эпизода».

Я считаю, что можно ясно изложить набор систематических методов для всех уровней построения программы и что студенты, курс обучения которых был построен таким образом, имеют значительное преимущество перед теми, кто обучался стандартным образом, только лишь изучая уже готовые программы.

Ниже дается несколько примеров того, чему мы можем учить.

Когда я знакоблю студентов с возможностями ввода, свойственными какому-либо языку программирования, то использую стандартную парадигму интерактивного ввода в форме макрокоманды, названной мною PROMT\_READ\_CHECK\_ECHO, которая осуществляет считывание до тех пор, пока вводимые данные удовлетворяют проверке на правильность, а затем осуществляет их эхопередачу в выходной файл. Эта макрокоманда является сама по себе на некотором уровне парадигмой итерации и ввода. В то же время, поскольку она чаще считывает один раз, чем выдает диагноз «неверные данные», она вводит более общую, изученную ранее парадигму цикла, выполняемого « $n+1/2$  раз».

```
PROMPT_READ_CHECK_ECHO: аргументы являются цепочкой
PROMPT, переменная V, которая должна быть считана, и усло-
                           вие BAD, характеризующее неверные данные;
PRINT_ON_TERMINAL (PROMPT);
READ_FROM_TERMINAL (V);
WHILE BAD (V) DO
  BEGIN
    PRINT_ON_TERMINAL («НЕВЕРНЫЕ ДАННЫЕ»),
    READ_FROM_TERMINAL (V)
  END;
PRINT_ON_FILE (V)
```

Она также, на более высоком уровне, иллюстрирует ответственность программиста по отношению к пользователю данной программы, включая следующий принцип: каждый компонент программы должен быть защищен от ввода, для которого он не предназначался.

Говард Шрауб и другие члены Группы практического обучения программированию [22] в Массачусетском технологическом институте успешно обучают своих студентов-новичков парадигме, полезной во многих случаях, которую они называют «создавать/фильтровать/накапливать». Студенты учатся распознавать многие внешне несхожие задачи как состоящие из перечисления элементов некоего множества, отфильтровывания подмножества и накапливания некой функции от элементов этого подмножества.

Язык MACLISP [18], используемый студентами, поддерживает эту парадигму, студенты создают только генератор, фильтр и накапливающий сумматор.

Ранее упоминавшаяся мною модель популяционной динамики хищник—жертва также представляет собой пример общей парадигмы, — парадигмы конечных автоматов. Обычно в парадигме конечных автоматов состояние вычисления представляется с помощью множества хранимых переменных. Если состояние сложное, то функция переходов требует создания механизма, способного поддерживать парадигму одновременного присваивания, особенно в силу того, что большинство языков лишь слабо поддерживают одновременное присваивание. Чтобы проиллюстрировать это, предположим, что мы хотим вычислить

$$\frac{\pi}{6} = \arcsin\left(\frac{1}{2}\right) = \frac{\boxed{1}}{\boxed{2} \cdot \boxed{1}} + \frac{\boxed{1}}{\boxed{2^3 \cdot 2} \cdot \boxed{3}} + \frac{\boxed{1 \cdot 3}}{\boxed{2^5 \cdot 2 \cdot 4} \cdot \boxed{5}} + \frac{\boxed{1 \cdot 3 \cdot 5}}{\boxed{2^7 \cdot 2 \cdot 4 \cdot 6} \cdot \boxed{7}} + \dots$$

где в каждом из слагаемых заключены в рамочку части, которые окажутся полезными при вычислении следующего члена, стоящего правее. Без описания всей парадигмы построения для таких процессов часть построения смены состояния заключается в постоянном нахождении способа перейти от

$$Q = \frac{1 \cdot 3}{2^5 \cdot 2 \cdot 4}, \quad C = 5$$

$$S = \frac{1}{2} + \frac{1}{2^3 \cdot 2 \cdot 3} + \frac{1 \cdot 3}{2^5 \cdot 2 \cdot 4 \cdot 5}$$

К

$$Q' = \frac{1 \cdot 3 \cdot 5}{2^7 \cdot 2 \cdot 4 \cdot 6}, \quad C' = ?$$

$$S' = \frac{1}{2} + \dots + \frac{1 \cdot 3 \cdot 5}{2^7 \cdot 2 \cdot 4 \cdot 6 \cdot 7}.$$

Опытный программист усвоил этот шаг, и во всех, за исключением самых сложных, случаях автоматически прибегает к нему. Новичку явная демонстрация парадигмы позволяет решать более сложные задачи теории конечных автоматов, чем он мог бы без этой помощи, и, что еще более важно, поощряет его самостоятельно находить другие полезные парадигмы.

Большинство классических алгоритмов, помещаемых в учебниках по программированию, могут рассматриваться в качестве примеров более широких парадигм. Формула Симпсона является примером экстраполяции к пределу. Метод исключения Гаусса представляет собой решение задачи методом рекурсивного спуска, преобразованным в итеративную форму. Сортировка слиянием является примером парадигмы «разделяй и властвуй». В отношении каждого такого классического алгоритма можно задать вопрос: «Каким образом я мог бы придумать его?» и восстановить, какой должна быть эквивалентная классическая парадигма.

В итоге мое обращение к серьезным программистам состоит в следующем: отводите часть вашего рабочего дня анализу и уточнению своих собственных методов. Даже несмотря на то, что программистам всегда приходится укладываться в сроки, которые уже либо прошли, либо приближаются, методологическая абстракция является разумным долгосрочным капиталовложением.

Преподавателю программирования я могу сказать даже больше этого: определите используемые вами парадигмы настолько полно, насколько вы можете, и затем ясно излагайте их. Они пригодятся вашим студентам, когда Фортран станет образцом мертвого языка-прототипа вроде латинского или санскрита.

Разработчику языков программирования я скажу: пока вы не можете поддерживать парадигмы, которые я использую при написании программ, или по меньшей мере поддерживать *мое* расширение вашего языка до такого, который действительно поддерживает мои методы программирования, я не нуждаюсь в ваших блестящих новых языках; как старый автомобиль или дом, старый язык обладает ограничениями, с которыми я научился уживаться. Чтобы убедить меня в достоинствах вашего языка, вы обязаны продемонстрировать мне, как на нем



писать программы. Я не хочу отбивать охоту к созданию новых языков, я хочу подтолкнуть разработчика языков к серьезному изучению деталей процесса разработки.

Я благодарю членов АСМ за то, что они включили меня в число знаменитых людей — моих предшественников по чтению тьюринговских лекций. Никто не может достичь такого положения без помощи других. Я должен выразить свою признательность многим, но особенно это относится к четверым: Бену Миттмэну, который на начальном этапе моей карьеры помогал мне и поощрял научную и преподавательскую стороны моего интереса к информатике; Гербу Саймону — человеку эпохи Ренессанса в нашей профессии, беседы с которым представляли собой подлинную школу; Джорджу Форсайту, снабдившему меня парадигмой обучения информатике, и моему коллеге Дональду Кнуту, создавшему общепризнанный образец интеллектуальной целостности. Мне также сопутствовала удача и в том отношении, что у меня было много прекрасных аспирантов, от которых, как мне кажется, я почерпнул столько же, сколько и дал им.

Я благодарен и глубоко признателен всем вам.

#### ЛИТЕРАТУРА

1. Aho A. V., Hopcroft J. E., and Ullman J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974. [Имеется перевод: Ахо А., Хопкрофт Дж., Ульман Дж., Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979.]
2. Aho A. V. and Ullman J. D. The Theory of Parsing, Translation, and Compiling. Vol. 1: Parsing Prentice-Hall, Englewood Cliffs, New Jersey, 1972. [Имеется перевод: Ахо А., Ульман Дж., Теория синтаксического анализа и трансляции. Т. 1. — М.: Мир, 1978.]
3. Balzer R. Imprecise Program Specification. Report ISI/RR—75—36, Inform. Sciences Inst., Dec. 1975.
4. Conway M. E. Design of a separable transition-diagram compiler Comm. ACM 6, 7 (July 1963), 396—408.
5. Davis R. Interactive transfer of expertise: Acquisition of new inference rules. Proc. Int. Joint. Conf. of Artif. Intell. MIT, Cambridge, Mass., August 1977, pp. 321—328.
6. Dijkstra E. W. Notes on structured programming. In Structured Programming, O. J. Dahl, T. W. Dijkstra and C. A. R. Hoare, Academic Press, New York, 1972, pp. 1—82. [Имеется перевод: Дал У., Дейстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975.]
7. Donzeau-Gouge V., Huet G., Kahn G., Lang B., and Levy J. J. A structure oriented program editor: A first step towards computer assisted programming. Res. Rep. 114, IRIA, Paris, April 1975.
8. Floyd R. W. The syntax of programming languages—A survey. IEEE ES—13. 4 (Aug. 1964), 346—353.
9. Floyd R. W. Nondeterministic algorithms. J. ACM 14, 4 (Oct. 1967), 636—644.
0. Gelernter. Realisation of a geometry-theorem proving machine. In Computers and Thought, E. Feigenbaum and J. Feldman, Eds., McGraw-Hill, New York, 1963, pp. 134—152.

11. Green C. C. and Barstow D. On program synthesis knowledge. *Artif. Intel.* 10, 3 (June 1978), 241—279.
12. Hewitt C. PLANNER: A language for proving theorems in robots. *Proc. Int. Joint Conf. on Artif. Intel.*, Washington D. C. 1969.
13. Hewitt C. Description and theoretical analysis (using schemata) of PLANNER... AI TR-258, MIT, Cambridge, Mass., April 1972.
14. Hoare C. A. R. Communicating sequential processes. *Comm. ACM* 21, 8 (Aug. 1978), 666—677.
15. Jensen K. and Wirth N. *Pascal User Manual and Report*. Springer-Verlag. New York, 1978. [Имеется перевод: Йенсен К., Вирт Н., Паскаль. Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1982.]
16. Kuhn T. S. *The Structure of Scientific Revolutions*. Univ. of Chicago Press, Chicago, Ill., 1970. [Имеется перевод: Кун Т. Структура научных революций. — М.: Прогресс, 1977.]
17. Lawler T., and Wood D. Branch and bound methods: A survey. *Operations Res.* 14, 4 (July—Aug. 1966), 699—719.
18. MACLISP Manual. MIT, Cambridge, Mass., July 1978.
19. Minsky M. Form and content in computer science. *Comm. ACM* 17, 2 (April 1970), 197—215.
20. Nilsson N. J. *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971. [Имеется перевод: Нильсон Н. Искусственный интеллект. Методы поиска решений. — М.: Мир, 1973.]
21. Parnas D. On the criteria for decomposing systems into modules. *Comm. ACM* 15, 12 (Dec. 1972), 1053—1058.
22. Rich C. and Shrobe H. Initial Report on a LISP programmer's apprentice. *IEEE J. Software Eng.* SE-4, 6 (Nov. 1978), 456—467.
23. Rulifson J. F., Derkson J. A. and Waldinger R. J. QA4: A procedural calculus for intuitive reasoning. Tech. Note 73, Stanford Res. Inst. Menlo Park, Calif., Nov. 1972.
24. Shortliffe E. H. *Computer-Based Medical Consultations: MYCIN*. American Elsevier, New York, 1976.
25. Sussman G. J., Winograd T. and Charniak C. *MICROPLANNER reference manual*. AI Memo 203A, MIT, Cambridge, Mass., 1972.
26. Teitelman W., et al. *INTERLISP manual*. Xerox Palo Alto Res. Ctr., 1974.
27. Wirth N. Program development by stepwise refinement. *Comm. ACM* 14 (April 1971), 221—227.
28. Wirth N. The programming language Pascal. *Acta Informatica* 1, 1 (1971), 35—63.
29. Wirth N. *Systematic Programming, an Introduction*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973. [Имеется перевод: Вирт Н., Систематическое программирование. Введение. — М.: Мир, 1977.]

1980

## Старые платья императора

*Чарлз Энтони Ричард Хоар*

Оксфордский университет, Англия

27 октября 1980 г. на ежегодной конференции в Нэшвилле, штат Теннесси, председатель комитета по премиям Уолтер Карлсон вручил премию АСМ им. Тьюринга Чарлзу Энтони Ричарду Хоару, профессору вычислительной математики Оксфордского университета в Англии.

Профессор Хоар был избран Комитетом по премиям за свой фундаментальный вклад в определение и создание языков программирования. Его труд отличает неординарное сочетание интуиции, самобытности, элегантности и влияния на развитие информатики. Наибольшую известность он завоевал благодаря работе над аксиоматическими определениями языков программирования при помощи методов, которые принято называть аксиоматическими семантическими методами. Он разработал искусные алгоритмы, такие как алгоритм Быстрой сортировки, а также изобрел и пропагандировал современные методы структуры данных в научных языках программирования. Он также внес важный вклад в развитие операционных систем в результате изучения мониторов. Самая последняя его работа посвящена последовательным взаимодействующим процессам.

До назначения в Оксфордский университет в 1977 г. профессор Хоар был с 1968 по 1977 г. профессором вычислительной математики в Королевском университете в Белфасте, Ирландия, и читал лекции в Станфордском университете в 1973 г. С 1960 по 1968 г. он работал на нескольких должностях в фирме Эллиотт Бразерс, Лимитед, Англия.

Профессор Хоар много публиковался и входит в состав редколлегии нескольких всемирно известных журналов по информатике. В 1973 г. ему присудили премию АСМ за лучшую работу по языкам программирования и программным системам. В 1978 г. профессор Хоар стал почетным членом Британского вычислительного общества, и ему присудили степень почетного доктора в университете Южной Калифорнии в 1979 г.

Тьюринговская премия — это наивысшая премия Ассоциации вычислительных машин за большой научный вклад в деятельность вычислительного сообщества. Она вручается каждый год.

в память д-ра А. М. Тьюринга, английского математика, внесшего большой вклад в развитие информатики.

Автор делится своим опытом в проведении в жизнь, разработке и стандартизации компьютерных языков программирования и делает одно предостережение на будущее.

\* \* \*

Мой первейший и самый приятный долг в этой лекции — выразить глубокую благодарность Ассоциации вычислительных машин за большую честь, оказанную мне, и за возможность обратиться к вам с докладом на любую тему по моему выбору. Но какой это трудный выбор! Мои научные достижения, столь полно оцененные этой премией, были уже достаточно описаны в научной литературе. Вместо того чтобы повторять малопонятную специальную терминологию из моей профессиональной области, я хотел бы немного неофициально поговорить о себе, о своем опыте, надеждах и опасениях, скромных успехах и менее скромных неудачах. Из неудач я почерпнул гораздо больше, чем можно рассказать сухим слогом научных статей, и теперь я хочу, чтобы вы тоже кое-чему научились на этих неудачах. Кстати, неудачи выглядят более забавными потом, когда о них рассказываешь, но когда они случаются, они не так уж забавны.

Начало моего рассказа относится к 1960 г., когда я стал программистом в небольшой компьютерной фирме, филиале компании Эллиотт Бразерс, Лимитед (Лондон), где в следующие за этим 8 лет я и должен был получить свое первоначальное компьютерное образование. Моим первым заданием было модифицировать для нового компьютера Эллиотт 803 библиотечную программу нового быстрого метода внутренней сортировки, как раз тогда изобретенного Шеллом. Я с радостью воспринял это как вызов максимизировать эффективность простой программы для десятично-адресуемой машины тех дней. Моему начальнику и наставнику Пэту Шеклtonу очень понравилась составленная мною программа. Тогда я застенчиво сказал, что мне кажется, будто я изобрел метод сортировки, который будет работать быстрее метода Шелла, и при этом он потребует ненамного больше памяти. Он поспорил со мной на шестипенсовик, что я ошибаюсь. Хотя мой метод трудно поддавался объяснению, все же мы пришли к выводу, что я выиграл пари.

Я написал несколько других высококачественных библиотечных подпрограмм, но через полгода мне было дано более важное задание — разработать новый усовершенствованный язык программирования высокого уровня для следующего компьютера компании Эллиотт 503, который должен был, имея тот же набор команд, что и предыдущая модель 803, быть в 60 раз

более производительным. Несмотря на то что я получил классическое образование и изучал древние языки, это было задание, для которого я обладал даже меньшей квалификацией, чем те, кто берется за эту работу сегодня. По счастью случайности в мои руки попал экземпляр отчета о международном алгоритмическом языке Алгол-60. Разумеется, этот язык был слишком сложным для наших клиентов. Как могли они разобраться во всех *begins* и *ends*, если даже наши продавцы не могли?

Примерно на Пасху 1961 г. в Брайтоне (Англия) был введен курс Алгола-60, который преподавали Питер Наур, Эдсгер Дейкстра и Питер Ланден. Я посещал этот курс вместе с моей сотрудницей по языковому проекту Джилл Пим, научным руководителем отдела Роджером Куком и коммерческим директором Полом Кингом. Именно здесь я впервые узнал о рекурсивных процедурах и увидел, как программировать метод сортировки, который я раньше считал трудным для объяснения. Именно здесь я написал программу, нескромно названную «быстрая сортировка (Quick Sort)», которая легла в основу моей карьеры ученого в области компьютеров. Следует отдать должное гению разработчиков Алгола-60 за то, что они включили в свой язык рекурсию и дали мне тем самым возможность весьма элегантно описать мое изобретение. Сделать возможным изящное выражение хороших мыслей — я считал это высшей целью проекта языка программирования.

По завершении курса Алгола-60 в Брайтоне, когда Роджер Кук вез нас с коллегами обратно в Лондон, он вдруг спросил: «А что если вместо разработки нового языка мы просто реализуем Алгол-60?» Мы все тут же с этим согласились — я теперь оцениваю это как счастливое для меня решение. Но мы знали в то время, что нам не хватит опыта и сноровки реализовать весь язык, и потому мне предстояло реализовать скромное его подмножество. В этом проекте я принял некоторые основные принципы, которые, я уверен, не утратили своей правильности и по сей день.

(1) Первым принципом была *надежность*. Согласно этому принципу, каждая синтаксически неправильная программа должна быть отброшена компилятором, а на каждую синтаксически корректную программу должен быть выдан результат или сообщение об ошибке, которое должно быть предсказуемым и понятным в терминах текста исходной программы. Следовательно, дампы памяти становились совершенно ненужными. Должно быть логически невозможно для любой программы на входном языке заставить компьютер выйти из-под контроля как во время компиляции, так и во время выполнения. Следствием этого принципа является то, что каждое вхождение каждого индекса каждой индексированной переменной в

каждом случае контролируется на выход за объявленные верхнюю и нижнюю границы массива. Много лет спустя мы спросили наших пользователей, хотели ли бы они, чтобы мы предусмотрели опцию транслятора для отключения всех этих проверок с целью повышения скорости выполнения программы. Они единодушно стали нас убеждать, чтобы мы этого не делали — они уже знали, насколько часто происходят ошибки с индексами во время выполнения программ, когда невозможность обнаружить их может оказаться роковой. Я отмечаю с ужасом, что даже в 1980 г. для разработчиков языков и пользователей это не послужило уроком. В любой достойной уважения отрасли техники невыполнение таких элементарных предосторожностей было бы давно уже признано нарушением закона.

(2) Второй принцип в разработке — *краткость объектного кода, порожденного компилятором, и компактность рабочих данных в ходе выполнения*. Причины этого ясны: размеры оперативной памяти любого компьютера ограничены и их расширение предполагает расходы и задержки. Программа, превышающая пределы памяти даже на одно слово, не может работать, в частности, потому, что многие из наших пользователей не намеревались приобретать дополнительные запоминающие устройства.

Принцип компактности объектного кода играет не меньшую роль и сейчас, когда процессоры тривиально дешевы по сравнению со стоимостью оперативного запоминающего устройства, к которому они могут обращаться, а дополнительные запоминающие устройства сравнительно даже еще дороже и медленнее на несколько порядков. Если в результате продуманной реализации доступная аппаратура окажется более мощной, чем может показаться необходимым для конкретного приложения, программист-прикладник почти всегда может воспользоваться дополнительными мощностями для повышения качества его программы, ее простоты, устойчивости и надежности.

(3) Третий принцип при разработке заключается в том, что *входные и выходные соглашения для процедур и функций должны быть столь же компактными и эффективными, как и для написанных в машинном коде подпрограмм*. Я полагал, что процедуры — это одно из самых мощных средств языка высокого уровня, так как они могут одновременно упростить работу программиста и сократить объектный код. Таким образом, не должно быть препятствий для их частого использования.

(4) Четвертый принцип состоял в том, что *компилятор должен быть однопроходным*. Компилятор состоял из набора взаимно рекурсивных процедур, каждая из которых могла анализировать и транслировать основные синтаксические едини-

цы языка — оператор, выражение, объявление и т. д. Он был разработан и документирован в Алголе-60 и затем перекодирован в десятичном машинном коде с использованием явного стека для рекурсии. Без понятия рекурсии в Алголе-60, в то время весьма спорного, мы не смогли бы написать этот компилятор.

Я и теперь могу рекомендовать однопроходный нисходящий рекурсивный спуск и в качестве метода реализации, и в качестве принципа разработки языка программирования. Во-первых, мы, конечно, предполагаем, что программу будут читать люди. А люди предпочитают прочитывать один раз, за один проход. Во-вторых, для пользователей системы с разделением времени или системы персональных компьютеров интервал между вводом программы с помощью клавиатуры (или внесением поправок) и запуском этой программы является полностью непродуктивным. Этот интервал можно минимизировать с помощью быстродействующего однопроходного компилятора. И наконец, придание компилятору структуры, соответствующей синтаксису его входного языка, в большой степени способствует обеспечению его корректности. Если у нас нет полной уверенности в этом, мы никогда не можем доверять результатам какой бы то ни было из наших программ.

Чтобы соблюсти все четыре перечисленных принципа, я выбрал довольно ограниченное подмножество Алгола-60. По мере конструирования и реализации я постепенно обнаруживал методы ослабления этих ограничений, не наносящие урона моим основным принципам. Поэтому в конце работы мы смогли реализовать почти полную мощность этого языка, включая и рекурсию, хотя некоторые его особенности были выброшены, а другие ограничены.

В середине 1963 г. главным образом в результате работы Джилл Пим и Джеффа Холмора была готова первая версия нашего компилятора. Через несколько месяцев у нас стали возникать вопросы, пользуется ли кто-нибудь нашим языком, заметил ли кто-нибудь время от времени выпускаемые нами отчеты, содержащие усовершенствования. Только когда у пользователя возникли претензии, он обращался к нам, но у большинства пользователей претензий не было. Теперь наши пользователи перешли на более современные компьютеры и используют более модные языки, но многие из них признались мне в том, что они с нежностью вспоминают о системе Эллиотт Алгол. И нежность эта связана не просто с ностальгией, но с эффективностью, надежностью и удобством этой прежней алголовской системы.

В результате работы над Алголом в августе 1962 г. я был приглашен в новую рабочую группу 2.1 IFIP, образованную

для поддержки и развития Алгола. Первой главной целью этой группы была разработка такого подмножества этого языка, из которого были бы удалены его наименее удачные свойства. Даже в те дни, имея дело с таким простым языком, мы понимали, что подмножество должно быть усовершенствованием исходного языка. Я очень много ждал от возможности встретиться и поговорить со многими мудрыми создателями этого языка. Я был удивлен и шокирован горячностью и даже озлобленностью их споров. Похоже, что первоначальная разработка Алгола-60 не велась в духе бесстрастных поисков истины, как я склонен был предполагать, исходя из его качества.

Чтобы отдохнуть от утомительной и вызывающей разногласия работы по разработке этого подмножества, рабочая группа наметила однажды обсудить те свойства, которые следовало бы включить в следующую версию языка. Каждому члену группы было предложено внести усовершенствование, которое он считал наиболее важным. 11 октября 1963 г. я предложил перейти к обсуждению просьбы наших пользователей об ослаблении правила, существующего в Алголе-60, обязательно объявлять имена переменных, и о принятии некоторого разумного соглашения по умолчанию, как, например, в Фортране. К моему крайнему удивлению, это, казалось бы невинное, предложение было вежливо, но твердо отклонено: было подчеркнуто, что избыточность Алгола-60 — это его наилучшая защита от ошибок программирования и кодирования, обнаружение которых в уже работающей программе обойдется слишком дорого, а еще дороже — если они не будут обнаружены вовсе. История о том, как космический корабль «Маринер», запущенный на Венеру, потерялся из-за отсутствия обязательного объявления в Фортране, была обнародована значительно позже. В конце концов я убедился в необходимости разрабатывать программистские обозначения так, чтобы максимизировать число ошибок, которые невозможно сделать, а если они все же сделаны, то число ошибок, надежно обнаруживаемых в процессе компиляции. Возможно, это увеличило бы длину программы. Но это не имеет значения! Разве не привело бы нас в восторг, если бы добрая фея предложила вам взмахом своей волшебной палочки над вашей программой убрать все ошибки с одним только условием — вы должны переписать и ввести всю вашу программу три раза! Способ сделать программу короче заключается в использовании процедур, а не в опущении жизненно важной информации, содержащейся в явном объявлении имен переменных.

Среди других предложений для развития нового Алгола было такое: switch в Алголе-60 предлагалось заменить более общим средством, а именно массивом переменных, значениями



которых являются метки, и так, что программа могла бы менять значения этих переменных с помощью присваивания. Я был категорически против этой идеи, подобной присваиваемой операции GO TO в Фортране, потому что я нашел поразительное количество мудреных проблем в реализации даже простых меток и переключателей в Алголе-60. А в этом новом свойстве я усматриваю еще больше проблем, включая такую, как возврат управления назад в блок после того, как он был выполнен. Я к тому же начинал подозревать, что программы, использующие много меток, труднее понимать и отлаживать, а для программ, в которых присваиваются новые значения переменным типа «метка», это сделать еще труднее.

Мне пришло в голову, что подходящая нотация для замены переключателя в Алголе-60 должна быть основана на обозначении условного выражения Алгола-60, которое делает выбор между двумя альтернативными действиями в соответствии со значениями булевского выражения. Другими словами, я предложил обозначения для операторов типа case, которые позволяют выбрать между любым числом альтернатив в зависимости от значения некоторого целого выражения. Это было мое второе предложение в разработке языка. Я до сих пор им горжусь, потому что в сущности оно совсем не создает проблем ни для разработчика, ни для программиста, ни для человека, читающего программу. Ныне, когда прошло более 15 лет, это обозначение было включено в международный нормативный документ по языку — исключительно *короткий* период внедрения по сравнению с другими областями техники.

Но вернемся к моей работе в фирме Эллиотт. После неожиданного успеха нашего компилятора с Алгола все наши pomysлы занял более честолюбивый проект: создать операционную систему для более крупной конфигурации компьютера 503, с устройством для считывания с перфокарт, построчными печатающими устройствами, магнитными лентами и даже с дополнительным запоминающим устройством на магнитных сердечниках, вдвое более дешевым и вдвое более емким, чем оперативное запоминающее устройство, но в пятнадцать раз более медленным. Это то, что позже стало называться системой программного обеспечения Эллиотт 503 Марк II.

Она включала в себя следующие компоненты:

(1) Транслятор с языка ассемблера, на котором было написано все остальное программное обеспечение.

(2) Схему автоматического управления оверлейных режимов для кодов и данных, как для накопителей на магнитных лентах, так и для дополнительного запоминающего устройства на магнитных сердечниках. Эта схема должна была использоваться остальным программным обеспечением.

(3) Схему автоматической буферизации всех вводов и выводов на любом доступном периферийном устройстве — снова для использования всем программным обеспечением.

(4) Файловую систему на магнитной ленте с возможностью для редактирования и управления заданиями.

(5) Полностью новую реализацию Алгола-60, лишенную всех нестандартных ограничений, которые мы ввели в нашей первой реализации.

(6) Компилятор с версии Фортрана, распространенной в то время.

Я писал документы, описывающие относящиеся к делу понятия и средства, и мы рассылали их существующим и потенциальным пользователям. В начале работы в нашей команде было 15 программистов, был назначен срок поставки: примерно через полтора года, в марте 1965 г. После начала разработки программного обеспечения Марк II я был внезапно повышен в должности и назначен помощником главного инженера, отвечающего за развитие и разработку аппаратных и программных продуктов фирмы.

Хотя в качестве администратора я был все еще ответствен за программное обеспечение Марк II, я уделял ему меньше внимания, чем новым продуктам фирмы, и почти что прозевал тот момент, когда истек срок поставки этого продукта, а ничего еще не было готово. Программисты пересмотрели свои графики реализации, и в июне 1965 г. была установлена новая дата поставки — через три месяца. Не стоит и говорить, что и эта дата прошла, а результатов все не было. К этому моменту наши клиенты начали сердиться, и мое начальство предложило взять мне этот проект под свою ответственность. Я попросил старших программистов снова подготовить исправленные графики, и снова оказалось, что проект сможет быть закончен через три месяца. Я очень хотел в это верить, но все же не смог. Я перестал обращать внимание на графики и углубился в проект.

Оказалось, что мы не составили никаких общих планов распределения нашего самого ограниченного ресурса — оперативной памяти. Каждый программист рассчитывал, что это будет сделано автоматически — либо ассемблером, либо автоматической схемой поддержки оверлеев. Хуже того, мы даже не рассчитали, какое пространство занимает наше собственное программное обеспечение, которым уже была заполнена вся оперативная память компьютера, что не оставляло места для работы программ наших пользователей. Ограничения на длину аппаратных адресов не позволяли увеличить оперативную память.

Стало ясно, что исходные технические требования к програм-

мному обеспечению не могли быть удовлетворены, и их следовало коренным образом урезать. Опытные программисты и даже менеджеры были освобождены от других проектов. Мы решили прежде всего сосредоточиться на поставке нового компилятора для Алгола-60, что должно было по самым тщательным подсчетам занять еще четыре месяца. Я постарался внушить всем программистам, занятым в проекте, что это больше не предсказание, а обещание; если они посчитают, что не могут выполнить этого обещания, они сами отвечают за то, чтобы найти пути и способы для выполнения работы.

Реакция на этот вызов со стороны программистов была замечательной. Они работали дни и ночи, чтобы обеспечить завершение всех тех элементов программного обеспечения, которые требовались для компилирующей программы на Алголе. К нашему восторгу, они уложились в назначенные сроки; это был первый важный фрагмент рабочего программного обеспечения, произведенный компанией за двухлетний период. Но восторг наш длился недолго: оказалось, что компилятор не может быть поставлен. Его скорость компилирования не превышала двух литер в секунду, что выглядело очень невыигрышно в сравнении с уже существующими компиляторами, работавшими со скоростью тысячи литер в секунду. Мы тотчас же нашли причину: она заключалась в метании между оперативной памятью и расширением — дополнительным запоминающим устройством на магнитных сердечниках, которое работало в пятнадцать раз медленнее. Нетрудно было внести несколько простых усовершенствований, и через неделю скорость компилятора удвоилась до четырех литер в секунду. В следующие две недели, посвященные исследованиям и перепрограммированиям, скорость снова была удвоена до восьми литер в секунду. Нам уже было ясно, как за месяц добиться дальнейшего улучшения; но число требуемых перепрограммирований увеличивалось, а их эффективность уменьшалась; это был слишком длинный путь. Альтернатива, состоящая в увеличении размера оперативной памяти, так часто выбираемая впоследствии при неудачах подобного рода, была невозможна из-за ограничений на адресацию аппаратного характера.

Выхода не было: надо было отказываться от всего проекта разработки программного обеспечения Эллиотта 503 Марк II, а вместе с ним выбросить в корзину около тридцати человеко-лет программистской работы, что эквивалентно почти целому сроку активной деятельности человека за всю его жизнь, и я нес ответственность и как разработчик, и как менеджер за их напрасную трату.

Было созвано заседание всех наших клиентов, работающих на модели 503, и Роджер Кук, бывший тогда менеджером от-

дела вычислений, объяснил им, что они никогда не получат ни одной команды столь долго ожидаемого программного обеспечения. Он докладывал об этом спокойным ровным голосом, это не позволяло никому из пользователей его прерывать, шептаться на задних рядах или даже ерзать на своих местах. Я смотрел на него с восхищением, но не мог разделять его спокойствия. Во время ленча наши пользователи были столь любезны, что пытались утешить меня. Они давно уже поняли, что программное обеспечение, соответствующее исходным техническим условиям, никогда не может быть поставлено, а если бы и было, то клиенты не знали бы, как воспользоваться их столь изощренными свойствами, и, во всяком случае, много столь же крупных проектов было аннулировано до поставки. Теперь, глядя в прошлое, я верю, что нашим клиентам повезло, что ограничения, вносимые аппаратурой, уберегли их от неограниченного потока наших разработок в области программного обеспечения. В наши дни пользователи микропроцессоров пользуются подобной же защитой — но это ненадолго.

В то время я прочитал документы, описывающие концепцию и свойства новой операционной системы 360 и новый проект разделения времени, названный Мультикс. Эти документы по своей понятности, продуманности и сложности превышали все, что я мог себе представить даже в первой версии программного обеспечения модели 503 Марк II. Очевидно, ИБМ и МТИ обладали каким-то секретом успешной разработки и реализации программного обеспечения, о природе которого я даже не пытался догадываться. Лишь позже эти фирмы осознали, что им он также неизвестен.

Таким образом, я до сих пор не могу понять, как мне удалось навлечь такое несчастье на мою фирму. В то время я был убежден, что мои начальники собираются меня уволить. Но они намеревались меня наказать еще более строго. «Хорошо, Тони, — сказали они. — Вы втянули нас в такие неприятности, а теперь вы нас вытащите из них». «Но я не знаю, как» — протестовал я. «Ну тогда вам придется в этом разобраться». Они даже выразили уверенность, что я справлюсь. Я этой уверенности не разделял. Я был склонен отказаться. Из всех моих удач самая большая — то, что я этого не сделал.

Разумеется, компания делала все, что было в ее силах, чтобы мне помочь. Они освободили меня от забот об аппаратных разработках и сократили численность моих программистских групп. Каждый из моих менеджеров изложил свою собственную теорию, объясняющую причину неудач, и у всех были разные теории. В конце концов они пригласили в мой офис самого старшего менеджера, генерального директора корпорации, в которую входила наша компания, Эндрью Сент-Джонс-

тона. Я был удивлен тем, что он что-то слышал обо мне. «Знаете, что у вас там произошло? — закричал он, он всегда кричал. — Вы позволили вашим программистам делать вещи, которых вы сами не понимаете». Я смотрел на него с удивлением. Очевидно, он был совершенно незнаком с современным положением вещей. Как один человек может целиком понимать любой современный программный продукт, такой, например, как система Эллиотт 503 Марк II?

Позже я понял, что он был абсолютно прав: он правильно диагностировал сущность проблемы и посеял семена правильного решения.

У меня по-прежнему была группа приблизительно из сорока программистов, и нам необходимо было удержать доверие клиентов нашей новой модели и даже снова завоевать его для нашей старой модели. Но что мы должны были планировать, если мы знали только одно — что все наши предыдущие планы потерпели неудачу? Тем не менее 25 октября 1965 г. я устроил продолжавшееся весь день совещание всех своих старших программистов, чтобы выяснить все накопившиеся недопонимания между нами. Я до сих пор храню свои заметки об этом совещании. Прежде всего мы перечислили основные претензии наших клиентов: отмена продуктов, неспособность уложиться в сроки, чрезмерный объем программного обеспечения «...не оправданный полезностью предусматриваемых средств», крайне медленные программы, неумение учитывать обратную связь с пользователем. «Своевременный учет самых скромных требований наших клиентов принес бы не меньше дивидендов в смысле желательности пользователей, чем успех наших самых честолюбивых планов».

Затем мы перечислили наши претензии: недостаток машинного времени для тестирования программ, отсутствие четкого графика выделения машинного времени, нехватка подходящего периферийного оборудования, ненадежность аппаратуры, даже если она и доступна, рассредоточенность программистских кадров, нехватка оборудования для клавишного перфорирования программ, отсутствие четких сроков поставки аппаратуры, нехватка технического персонала для документирования, недостаточность знаний в области программного обеспечения за пределами группы программистов, вмешательство вышестоящих администраторов, навязывающих свои решения «...без полного понимания менее очевидных последствий этих решений», а также излишний оптимизм под давлением клиентов и отдела сбыта.

Но мы не стремились оправдать этими претензиями нашу неудачу. Например, мы согласились с тем, что долг программистов — повышение научного уровня своих администраторов и

других отделов компании путем «...представления необходимой информации в простой осязаемой форме». Надежда на то, что «...недостатки спецификаций оригинальной программы можно было бы компенсировать мастерством отдела технической документации... не оправдались; разработка программы и разработка ее спецификации должны производиться параллельно одним и тем же лицом, и должно существовать взаимодействие между этими этапами работы. Недостаточная ясность спецификации является вернейшим признаком недостатков в программе, которую она описывает, и эти две ошибки должны устраняться одновременно, до того как приступают к проекту». Жаль, что я не следовал этому совету в 1963 г.; хотелось бы, чтобы все мы следовали ему теперь.

Мои заметки об однодневном совещании в октябре 1965 г. включали в себя целый раздел, посвященный неудачам группы программного обеспечения; этот раздел может соперничать с наиболее униженной самокритикой ревизионистов времен китайской культурной революции. Нашей главной ошибкой было излишнее честолюбие. «Очевидно, что цели, которых мы пытались достичь, оказались намного выше наших возможностей». Мы потерпели также неудачу в прогнозе размеров и быстрого действия программ в оценке того, какие требуются усилия: неудачу в планировании координации и взаимодействия программ между собой; нашей ошибкой было то, что мы своевременно не обнаружили, что дела идут плохо. Обнаружились также ошибки в контроле за изменениями в программах, в документировании, связи с другими отделами, с нашим начальством и с клиентами. Мы не смогли дать четкие и неизменяемые определения того, что должен сделать каждый программист и руководитель проекта; стоит ли продолжать? Что больше всего удивляет, так это то, что большая команда весьма умных и знающих программистов смогла так тяжело и так долго работать над таким малообещающим проектом. Известно, что не следует доверять нам, умным программистам. Мы можем выдумать достаточно хорошие доводы, чтобы убедить самих себя и других в какой угодно чепухе. Особенно не верьте нам, когда мы обещаем повторить прежний успех в следующий раз, только увеличив и улучшив его.

Последний раздел нашего расследования неудач содержал критерии качества программного обеспечения. «В нынешней борьбе за то, чтобы выдать хоть какое-нибудь программное обеспечение, первой жертвой оказывается качество готового программного продукта. Качество программного обеспечения оценивается несколькими несовместимыми в полной мере критериями, которые надо тщательно уравновесить при разработке и реализации каждой программы». Затем мы составили список,

в который вошло не менее семнадцати критериев, который опубликовали в редакционной статье т. 2 журнала «Software Practice and Experience».

Как удалось нам встать на ноги после этой катастрофы? Во-первых, мы разбили клиентов модели 503 на несколько групп в соответствии с характером и объемом аппаратных конфигураций, которые они приобрели, — например, все приобретшие оборудование с магнитными лентами были отнесены к одной группе. К каждой группе пользователей мы прикрепили небольшую команду программистов и предложили руководителям команд посетить клиентов и разобраться, чего они хотят, выбрать требования, которые проще всего удовлетворить, а затем наметить планы (но не давая обещаний) их выполнения. Ни в коем случае нельзя рассматривать требования, удовлетворение которых заняло бы более трех месяцев. Руководитель проекта должен был убедить меня, что требование клиентов разумно, что разработка нового свойства нужна и что планы и график работ по реализации реалистичны. Самое главное, я не позволял делать ничего, чего бы сам не понимал. И это подействовало! Запросы на программное обеспечение стали выполняться вовремя. Благодаря увеличению доверия между нами и нашими клиентами мы смогли приступить к выполнению более сложных требований. Через год мы оправились от потрясения. Через два года у нас даже были умеренно удовлетворенные клиенты.

Таким образом, благодаря здравому смыслу и компромиссам мы добились чего-то похожего на успех. Но я не был удовлетворен. Я не мог понять, почему разработка и реализация операционной системы должны быть настолько труднее разработки и реализации компилятора. Именно поэтому я посвятил мои последующие исследования проблемам параллельного программирования и языковым конструкциям, которые должны были способствовать четкому структурированию операционных систем — таким, как мониторы и взаимодействующие процессы.

Когда я работал в компании Эллиотт, я сильно заинтересовался методами формального определения языков программирования. В то время Питер Ландин и Кристофер Стречи предложили определить язык программирования с помощью простой функциональной нотации, которая бы определила результат выполнения каждой команды на математически определенной абстрактной машине. Мне не очень нравилось это предложение, поскольку я чувствовал, что такое определение должно включать в себя ряд вполне произвольных решений способа представления, и в принципе оно должно быть ненамного проще, чем реализация языка на реальной машине. В качестве альтернативы я предложил, чтобы определение языка программи-

рования было формализовано в виде последовательности аксиом, описывающих желаемые свойства программ, написанных на этом языке. Мне казалось, что тщательно сформулированные аксиомы оставляют достаточно свободы для эффективной реализации языка на разных машинах и позволяют программисту доказать правильность его программ. Но как это сделать, мне было неясно. Я думал, что потребуются длительные исследования для разработки и применения необходимых методов, а самым лучшим местом для проведения таких исследований является университет, а не промышленность. Таким образом, я обратился на кафедру информатики Королевского университета Белфаста, где мне предстояло провести девять счастливых и продуктивных лет. В октябре 1968 г., когда я распаковывал свои бумаги в новом доме в Белфасте, я наткнулся на препринт малоизвестной статьи Боба Флойда, озаглавленной «Assigning Meanings to Programs» (Придание смысла программам). Вот это была удача! Наконец-то я увидел, как достигнуть того, к чему я стремился в моих исследованиях. Тогда я написал свою первую статью об аксиоматическом подходе к программированию на компьютере, которая была опубликована в Communications of the ASM в октябре 1969 г.

Как раз незадолго до этого я обнаружил, что одним из первых, кто отстаивал метод проверки программы с помощью утверждений, был никто иной, как сам Алан Тьюринг. В июне 1950 г. на конференции в Кембридже он прочел короткое сообщение, озаглавленное «Проверка большой программы», в которой эта идея очень ясно объяснялась. «Как можно проверить большую программу, чтобы убедиться, что она верна? Чтобы облегчить задачу проверяющего, программист должен сделать некоторое число определенных *утверждений*, которые можно было бы проверить по отдельности и из правильности которых следовала бы правильность всей программы».

Рассмотрим аналогичный случай — проверку сложения. Если дана сумма (столбец цифр с ответом внизу), то надо проверять все за один раз. Если же даны суммы для нескольких столбцов (просуммированных по отдельности), работа проверяющего облегчается, поскольку она разбивается на проверки нескольких различных утверждений (т. е. что каждый столбец правильно просуммирован) и небольшое суммирование (чтобы получить итоговую сумму). Этот принцип можно применить к проверке большой программы, но мы проиллюстрируем этот метод маленькой программой, а именно получение  $n$ -факториала без помощи устройства умножения. К несчастью, не существует достаточно широко известной системы кодирования, чтобы можно было оправдать приведение здесь всей программы, но для иллюстрации достаточно блок-схемы. Это возвращает



меня к основной теме моего доклада, к разработке языков программирования.

С августа 1962 по октябрь 1966 г. я бывал на каждом собрании рабочей группы по Алголу при IFIP. После завершения нашей работы по подмножеству Алгола IFIP мы начали разрабатывать Алгол-X, который намеревались сделать преемником Алгола-60. Были выдвинуты дополнительные предложения о включении в язык новых свойств, и в мае 1965 г. Никлаусу Вирту было поручено собрать их все и разработать единый проект языка. Я пришел в восторг от его чернового проекта, которому удалось избежать всех известных недостатков Алгола-60 и который содержал несколько новых свойств. Все они могли быть легко и эффективно реализованы, а их применение было надежным и удобным.

Описание этого языка все еще не было закончено. Я много поработал над его улучшением, и многие члены нашей бригады тоже. Ко времени следующего совещания в октябре 1965 г. во Франции, в местечке Сент-Пьер де Шартрез, у нас был черновой вариант замечательного и реалистичного проекта языка, который был опубликован в июне 1966 г. под названием «Вклад в развитие Алгола» в Communications of the ACM. Он был реализован на IBM-360 и получил название Алгол-W, которое ему дали его многочисленные счастливые пользователи. Это был не только удачный преемник Алгола-60, это был к тому же удачный предшественник Паскаля.

На том же самом собрании комитету по Алголу был предложен краткий, неполный и довольно невразумительный документ, описывающий другой, более претенциозный и, по моему мнению, куда менее привлекательный язык. Я был поражен, когда рабочая группа, состоящая из всех наиболее известных международных экспертов по языкам программирования, решила оставить в стороне черновой проект, над которым мы все работали, и заглотнула такую неаппетитную приманку.

Это произошло как раз через неделю после нашего заключения о проекте программного обеспечения 503 Марк II. Я выразил отчаянное предостережение против нечеткости, сложности и чрезмерной претенциозности нового проекта, однако мой голос не был услышан. Я пришел к заключению, что существуют два способа составления проекта программного обеспечения: один способ — сделать его таким простым, чтобы было *очевидно*, что недостатков нет, а другой — сделать его таким сложным, чтобы не было *очевидных* недостатков.

Первый метод намного труднее. Он требует такой же смелости, преданности делу, проникаемости и даже вдохновения, как открытие простых физических законов, лежащих в основе сложных явлений природы. Это также требует готовности при-

нять цели, ограниченные физическими, логическими и технологическими требованиями, и согласиться на компромисс, когда нельзя достигнуть согласия между противоречивыми требованиями. Ни один комитет не сможет добиться этого своевременно, согласие достигается лишь тогда, когда все сроки уже прошли.

Именно это и случилось с комитетом по Алголу. Было ясно, что проект, который они предпочли, не был совершенным. Поэтому было обещано, что новый и окончательный проект нового языка Алгол будет готов через три месяца. Он должен был быть затем тщательно изучен подгруппой из четырех членов группы, в которую входил и я. Три месяца прошли, но ни слова о новом проекте не было слышно. Через полгода подгруппа устроила совещание в Нидерландах. У нас был более длинный и более толстый документ, полный исправленных в последнюю минуту ошибок, описывающий еще один новый, но для меня столь же непривлекательный язык. Никлаус Вирт и я провели некоторое время, пытаясь устранить некоторые недостатки в проекте и в описании, но тщетно. Завершенный окончательный набросок языка был обещан на следующей встрече всего комитета по Алголу, намеченной через три месяца.

Снова прошли три месяца — и ни слова о новом наброске не появилось. Через полгода, в октябре 1966 г., рабочая группа по Алголу собралась в Варшаве. Она рассматривала еще более толстый и длинный документ, пестрящий исправленными в последнюю минуту ошибками, описывающий еще один, столь же темный, а для меня столь же непривлекательный язык. Эксперты группы не смогли увидеть недостатков проекта и твердо решили принять набросок в надежде, что он будет закончен через три месяца. Напрасно я говорил им, что он не будет готов. Напрасно я настаивал на устранении технических ошибок языка, преобладания ссылок, преобразований типа по умолчанию. Рабочая группа отнюдь не стремилась к упрощению языка и требовала от авторов, чтобы они включили еще более сложные свойства, такие, как совмещение операторов и параллелизм.

Когда какой-нибудь проект нового языка близится к завершению, всегда возникает безумная спешка — внести новые свойства еще до стандартизации. Это стремление действительно безумно, потому что оно приводит в ловушку, из которой нет выхода. Недостающие свойства всегда можно добавить позже, когда их конструкция и их последствия будут хорошо поняты. Свойство, включенное прежде, чем оно было понято, никогда нельзя изъять позже.

Наконец в декабре 1968 г., настроенный крайне пессимистически, я поехал на встречу в Мюнхен, где наш долго вынашиваемый монстр должен был появиться на свет и получить

ия Алгол-68. К тому времени некоторые другие члены группы также разочаровались в проекте, но было слишком поздно: теперь комитет был заполнен сторонниками этого языка и проект был направлен для утверждения вышестоящим органам IFIP. Все, что мы смогли сделать, — отправить вместе с ним заявление меньшинства, формулирующее наше твердое убеждение, что «... как инструмент для надежного создания сложных программ данный язык непригоден». Этот отчет был позднее скрыт — акт, который напомнил мне строчки из Хилэра Беллока:

Ученые мужи — не верить им грешно!  
Нас уверяют, что так быть должно.  
Не дай нам Бог сомнение допустить  
В том, в чем никто себя не в силах убедить<sup>1)</sup>.

Я больше не был ни на одном собрании этой рабочей группы. Я с удовольствием сообщаю, что вскоре эта группа пришла к пониманию, что с этим языком, а также с его описанием не все в порядке; они напряженно проработали еще шесть лет, чтобы создать пересмотренное описание этого языка. Это было значительным усовершенствованием, однако боюсь, что, на мой взгляд, оно не устранило основных технических изъянов проекта, а также полностью проигнорировало проблему чрезмерной сложности этого языка.

Программистам всегда приходится соприкасаться со сложностью; этого нельзя избежать. Наши приложения сложны, потому что мы честолюбивы и стремимся использовать наши компьютеры все более сложными способами. Программирование сложно из-за большого числа противоречивых целей, преследуемых каждым программным проектом. Если наш основной инструмент, язык, на котором мы составляем и кодируем программу, тоже непрост, то сам язык становится частью нашей проблемы, а не ее решения.

А теперь я расскажу вам о другом излишне амбициозном языковом проекте. Между 1965 и 1970 г. я состоял членом и даже председателем технического комитета № 10 Европейской ассоциации изготовителей компьютеров. Сначала нам было предложено рассмотреть краткое резюме, а затем стандартизировать некий язык, который должен был вытеснить все языки и был предназначен для всех приложений компьютеров, как научных, так и коммерческих; инициатива исходила от крупнейшего изготовителя компьютеров всех времен. С интересом и удивлением и даже с некоторым удовольствием я изучил четыре первоначальных документа, описывающие язык NPL, по-

---

<sup>1)</sup> Перевод С. В. Чудова. — *Прим. ред.*

явившиеся в период между мартом и ноябрем 1964 г. Из них каждый следующий был более претенциозным и абсурдным, чем предыдущий, в своих обещаниях и желаниях превзойти все известное ранее. Затем язык начали реализовывать, с интервалами в полгода появлялись все новые документы, и каждый из них описывал новый окончательный вариант этого языка под его окончательным названием PL/I.

Но, на мой взгляд, каждый пересмотр документа просто показывал, как далеко продвинулась реализация первоначального замысла. Те компоненты языка, которые еще не были реализованы, по-прежнему описывались свободной цветистой прозой, обещающей ничем не омраченное удовольствие. А в тех компонентах, которые уже были реализованы, цветы уже поблекли: их заглушила поросль объяснительных примечаний, накладывающих произвольные и неприятные ограничения на использование каждого свойства и налагающих на программиста ответственность за отслеживание сложных и неожиданных побочных эффектов и эффектов взаимодействия с другими свойствами языка.

Наконец, 11 марта 1968 г. описание языка было честь честью представлено с нетерпением его ожидающей научной общественности в качестве достойного кандидата на стандартизацию. Но он им не был. Это описание уже подверглось семи тысячам исправлений и модификаций его авторов и создателей. Понадобилось еще двенадцать изданий, прежде чем он был окончательно опубликован в качестве стандарта в 1976 г. Я боюсь, что это произошло не потому, что все причастные к этому проекту люди были удовлетворены своей работой, а потому, что им все это надоело и они расстались с иллюзиями.

Все то время, пока я хоть как-то участвовал в этом проекте, я не устал добиваться, чтобы язык был упрощен; если это необходимо, хотя бы выделением ограниченного его подмножества, которое профессиональный программист смог бы понять и отвечать за правильность, стоимость и эффективность своих программ. Я настаивал на том, чтобы опасные свойства, такие, как соглашения по умолчанию и ON-условия, были устранены. Я знал, что будет невозможно написать полностью надежный компилятор для столь сложного языка, когда правильность каждой части программы зависит от проверки того, чтобы все другие части этой программы избежали всех ловушек и ошибок языка.

Вначале я надеялся, что такой технически неразумный проект потерпит крах, но вскоре я понял, что он обречен на успех. Почти все в программном обеспечении может быть реализовано, продано и даже использовано, если проявить достаточную настойчивость. Ничто из того, что может утверждать какой-то

там ученый, не может остановить поток сотен миллионов долларов. Но существует одно качество, которое нельзя купить таким образом, — это надежность. Цена надежности — это погоня за крайней простотой. Это цена, которую очень богатому труднее всего заплатить.

Все это произошло много лет тому назад. Можно ли считать, что все это имеет отношение к конференции, посвященной предвидению компьютерной эры, на пороге которой мы стоим? Больше всего я боюсь, что имеет, и притом самое непосредственное. Ошибки, которые мы совершили за последние двадцать лет, продолжают повторять ныне и даже в больших масштабах. Я имею в виду проект разработки языка, который породил документы, озаглавленные «соломенный человек», «деревянный человек», «жестяной человек», «железный человек», «стальной человек», «зеленый» и, наконец, АДА. Этот проект был задуман и финансирован одной из самых могущественных организаций мира, Министерством обороны США. Таким образом, ему обеспечено влияние и внимание независимо от его технических достоинств, а его ошибки и дефекты угрожают нам гораздо большими опасностями. Ни одно из имеющихся пока свидетельств не может внушить нам уверенность в том, что в этом языке удалось избежать тех проблем, с которыми сталкивались другие сложные языковые проекты прошлого.

Я высказывал все, какие мог, советы по этому проекту начиная с 1975 г. Вначале я был очень оптимистичен. Первоначальные цели разработки включали надежность, читаемость программ, формализованность определения языка и даже простоту. Постепенно эти цели приносились в жертву мощности, якобы достигнутой благодаря обилию свойств и соглашений по обозначениям, многие из которых были необязательными, а некоторые из них, такие, как обработка исключительных ситуаций, даже опасными. Обратимся к истории конструирования автомобиля. Мелкие и не очень нужные приспособления преобладают над соображениями безопасности и экономичности.

Но еще не поздно! Я верю, что с помощью старательного удаления лишнего из языка АДА все еще возможно выбрать очень мощное подмножество, которое было бы надежно и эффективно при реализации, а также безопасно и экономично в использовании. Спонсоры этого языка высказывались недвусмысленно, что никаких подмножеств не будет. Это самый странный парадокс этого странного проекта. Если вы хотите, чтобы у языка не было подмножеств, создавайте маленький язык.

Вы включаете только те свойства, о которых вы знаете, что они необходимы для *каждого* приложения языка, и о которых вы знаете, что они годятся для *каждой* аппаратной конфигура-

ции, на которой этот язык реализован. Тогда там, где необходимо, должны быть разработаны расширения для конкретных аппаратных устройств и для конкретных приложений. Великая сила Паскаля в том и состоит, что в нем очень мало ненужных свойств и почти нет нужды в подмножествах. Вот почему этот язык достаточно силен, чтобы выдержать специализированные расширения — Параллельный Паскаль для работы в реальном времени, Паскаль-плюс для моделирования дискретных событий и UCSD-Паскаль для микропроцессорных рабочих станций. Если бы только мы могли извлекать правильные уроки из прошлых успехов, нам не было бы нужды учиться на наших неудачах.

Итак, лучший из моих советов организаторам и разработчикам языка АДА остался без внимания. И вот, как последнее средство, я обращаюсь к вам, представителям программистской общественности США, а также гражданам, причастным к благосостоянию и безопасности вашей страны и всего человечества: не позволяйте этот язык в его теперешнем состоянии использовать в приложениях, надежность которых критична, таких, как атомные электростанции, крылатые ракеты, системы раннего оповещения, системы противоракетной обороны. Следующая ракета, которая сойдет к пути в результате ошибки в языке программирования, может оказаться не исследовательским космическим зондом, летящим в безобидное путешествие к Венере; это может быть ракета с ядерной боеголовкой, которая способна взорваться над одним из наших собственных городов. Ненадежный язык программирования, порождающий ненадежные программы, представляют для окружающей среды и нашего общества несравненно больший риск, чем небезопасные автомобили, токсические пестициды или аварии на атомной электростанции. Неусыпно добивайтесь снижения риска, а не увеличения его.

Позвольте мне закончить не на такой мрачной ноте. Видеть, как твои лучшие советы игнорируются, — такова судьба всех, кто берет на себя роль консультанта, еще с тех пор, когда Кассандра предупредила, что опасно ввозить деревянного коня внутрь стен Трои. Это напомнило мне одну историю, которую я слышал в детстве. Насколько я помню, она называлась так:

### Старые платья императора

Много лет тому назад жил один император, который так любил одеваться, что тратил все свои деньги на платья. Он не интересовался армией, не устраивал пиров, не вершил судьбы в суде. О других королях или императорах можно было сказать: «Он заседает в совете», но об этом всегда говорили: «Император сидит в своем гардеробе». Так оно и было. В один не-

счастный день он был обманут и вышел к народу голым — к своему огорчению и к восторгу своих поданных. Он решил никогда не покидать свой трон и, чтобы не показаться снова перед народом голым, приказал, чтобы каждое из его многочисленных новых одеяний просто надевалось сверху на прежнее.

Время проходило весело в его огромном столичном городе. Министры и придворные, ткачи и портные, гости и поданные, швеи и вышивальщицы заходили и выходили из тронного зала, каждый по своим делам, и все они восклицали: «Как прекрасно одеяние нашего императора!»

Однажды старейший и самый верный министр императора услышал об искусном портном, который закончил старейшую высшую школу швейного мастерства и разработал новое искусство абстрактной вышивки, в которой использовались столь тонкие стежки, что никто не мог с уверенностью сказать, были ли они или нет. «Должно быть, это действительно замечательные стежки, — подумал министр. — Если бы мы только смогли пригласить этого портного в советники, мы достигли бы в украшении нашего императора таких высот восхваления, что весь мир признал бы его величайшим из всех императоров».

Итак, старый честный министр нанял портного за большое вознаграждение. Портного привели в тронный зал, и он сделал почтительный поклон груди одежды, которая теперь полностью покрывала весь трон. Все придворные с нетерпением ждали его советов. Представьте себе их изумление, когда он посоветовал не добавлять изысканности и более сложной вышивки к ранее существующей, а посоветовал убрать слои пышных нарядов, стремиться к простоте и элегантности вместо экстравагантной утонченности. «Этот портной не такой уж знаток, как он утверждает, — бормотали они. — Его мозги протухли от долгого созерцания его башни из слоновой кости, и он больше не понимает портновских потребностей современного императора». Портной долго и громко отстаивал здравомыслие своего совета, но не мог добиться, чтобы его послушались. В конце концов он получил гонорар и вернулся в свою башню из слоновой кости.

Никогда, вплоть до этого самого дня, не была поведена вся правда о той истории. А она такова. Однажды утром, когда императору стало жарко и скучно, он старательно высвободился из-под груды одежд и зажил счастливо, как свинопас из другой истории. Портной был канонизирован как святой патрон всех консультантов, потому что, несмотря на огромный гонорар, который он получил, он так и не смог убедить своих клиентов в том, что он давно начал подозревать: императора в одеждах нет.

1983

## Размышления об исследованиях в области программного обеспечения

*Дэнис М. Ритчи*

AT&T Bell Laboratories

Премия Тьюринга 1983 г. была вручена в октябре на ежегодной конференции ACM Дэнису М. Ритчи и Кену Томпсону из AT&T Bell Laboratories за разработку и реализацию операционной системы UNIX.

Система разделения времени UNIX была задумана Томпсоном и реализована им совместно с Ритчи в конце 60-х годов. Ключевым вкладом в переносимость системы UNIX был созданный Ритчи язык программирования Си.

Их отправная статья «Система разделения времени UNIX» была первоначально представлена на Четвертом симпозиуме ACM по принципам операционных систем в 1973 г., а затем в пересмотренном виде появилась в выпуске «Communications of the ACM» за июль 1974 г. Эта статья получила приз ACM за лучшую работу по языкам программирования и системам в 1974 г.

Как сказано в решении Комитета по премиям Тьюринга, «Успех системы проистекает из тщательного выбора нескольких ключевых идей и элегантной их реализации. Пример системы UNIX привел поколение разработчиков программного обеспечения к переосмыслению основ программирования. Основной принцип системы UNIX заключен в ее подходе, который позволяет программистам опираться на работу других».

Эта премия — высочайшее признание Ассоциацией научного вклада в области информатики. Она учреждена в честь Алана М. Тьюринга — английского математика, внесшего значительный вклад в информатику.

Ритчи и Томпсон на конференции прочитали отдельные лекции. Ритчи сфокусировал внимание на рабочей атмосфере в фирме Bell Laboratories, сделавшей возможной создание UNIX. Томпсон рассмотрел вопрос о том, насколько можно доверять программному обеспечению и насколько — людям, создающим его.

Статья Томпсона начинается на с. 203.

Могут ли обстоятельства, сложившиеся в Bell Laboratories, благодаря которым вырос проект UNIX, повториться опять?



Операционная система UNIX внезапно стала новостью, хотя она и не нова. Она зародилась в 1969 г., когда Кен Томпсон открыл для себя малоиспользуемую машину PDP-7 и решил оснастить ее программной средой на свой вкус. Его работа скоро привлекла и меня. Я присоединился к этому предприятую, хотя многие идеи и основная часть работы по их воплощению принадлежали ему. Вскоре эту систему стали использовать наши коллеги по исследовательской группе в AT&T Bell Laboratories. С особым энтузиазмом критиковали и дорабатывали систему Джо Осана, Дуг Макилрой и Бол Морис. В 1971 г. мы получили PDP-11 и к концу этого года обеспечивали работу наших первых настоящих пользователей — трех машинисток, вводящих заявки на патенты. В 1973 г. система была переписана на языке Си. В этом же году она была впервые обнародована на Конференции по принципам операционных систем; итоговая статья [8] появилась в Communications of the ACM в следующем году.

С тех пор использование системы постоянно росло как внутри, так и вне Bell Laboratories. Была создана группа разработчиков для поддержки проекта внутри компании и выдано несколько лицензий на разработку экспериментальных версий вне компании.

Последней поставляемой экспериментальной версией была седьмая редакция системы, появившаяся в 1979 г. Позднее AT&T стала продавать System III, а в настоящее время предлагает System V; обе системы — продукт группы разработчиков. Все исследовательские версии поставлялись «как есть», без сопровождения. System V — сопровождаемый продукт для нескольких различных линий машин, в том числе с недавнего времени для машин ряда 3B, спроектированных и созданных в AT&T.

Система UNIX широко используется, и сейчас даже идет речь о возможном промышленном стандарте. Как она достигла такого успеха?

Конечно, UNIX имеет свои технические достоинства. Так как система и ее история были широко описаны в литературе [6, 7, 11], я не буду говорить об этих ее достоинствах, кроме одного: несмотря на свою внешнюю шероховатость, так красочно расписанную Доном Норманом в его статье в Datamation [4], и несмотря на свое богатство, UNIX — простая и последовательная система, в которой небольшое количество понятий и идей было доведено до логического завершения. Именно это качество более, чем все прочие, привлекало к ней множество сторонников.

Помимо технической стороны дела успеху UNIX способствовали некоторые социологические факторы. Во-первых, она

появилась в то время, когда стали доступны другие возможности, кроме больших вычислительных центров с центральным администрированием: 1970-е годы были десятилетием мини-компьютеров. Небольшие группы могли устанавливать свои собственные вычислительные средства. Поскольку они начинали все заново и поскольку программное обеспечение, поставляемое производителями мини-компьютеров, было в лучшем случае бездейным, а часто отвратительным, некоторые рискованные люди решили попробовать использовать новую заманчивую, хотя и не поддерживаемую операционную систему.

Во-вторых, UNIX сначала стала доступна на PDP-11, одной из наиболее успешных новых мини-ЭВМ, появившихся в 1970-х годах, и вскоре благодаря своей мобильности была перенесена на многие новые машины по мере появления последних. В то время, когда создавалась система UNIX, мы усиленно настаивали на покупке новой машины — либо PDP-10 фирмы DEC, либо Sigma 7 фирмы SDS (позднее XEROX). В ретроспективе ясно, что если бы мы преуспели в покупке такой машины, система UNIX все же могла бы быть написана, но обречена на увядание. Аналогично UNIX многим обязана системе Multics [5]; как я уже писал [6, 7], она затмила свою предшественницу прежде всего потому, что не требовала необычного аппаратного обеспечения, а не из-за каких-либо других качеств.

Наконец, UNIX повезло в том, что она имела необычно длинный инкубационный период. Большую часть этого времени (скажем, 1969—1979 гг.) система находилась под полным контролем своих создателей и использовалась ими. Разработка всех идей и программирование требовали времени, но даже несмотря на то, что система все еще развивалась, она использовалась как внутри Bell Laboratories, так и вне (по лицензии). Таким образом, мы ухитрились держать под своим контролем основные идеи, в то же время накапливая общественную поддержку полных энтузиазма, технически компетентных пользователей, приносящих свои идеи и программы в спокойной, общительной обстановке, при отсутствии конкуренции. Некоторые внешние пользователи — например, из Калифорнийского университета в Беркли — внесли существенный вклад. Наши пользователи образовали хоть и тонкий, но широкий слой и в самой компании, в университетах, в некоторых коммерческих и правительственных организациях. Благодаря этой сети ранних пользователей система заняла важное место на интеллектуальном, если не на коммерческом рынке.

В чем состоят научные исследования в области промышленной информатики? Некоторые люди считают, что первоначально система UNIX разрабатывалась как нелегальный про-

ект, «халтурка». Это не так. От работников исследовательских отделов ожидается, что они открывают или изобретают новые вещи, и хотя на начальном этапе наше аппаратное обеспечение было скудным, нас всегда поддерживало руководство. В то же время это несомненно не имело ничего общего с обычным исследовательским проектом. Нашей целью было создание удобной вычислительной среды для себя, и мы надеялись, что она понравится другим. Научно-исследовательский центр по информатике в Bell Laboratories, которому принадлежим мы с Томпсоном, занимается тремя широкими темами: теория, численный анализ, системы, языки и программное обеспечение. Хотя самостоятельная научная работа, результатом которой является, например, статья в научном журнале, не только не встречает препятствий, но и приветствуется, имеется сильное, хотя и на удивление незаметное давление, заставляющее думать о задачах, каким-либо боком затрагивающих нашу корпорацию. Это было так с тех пор, как я поступил в Bell Laboratories примерно 15 лет назад, и это не должно удивлять: может показаться, что старая Bell System была протекционистской монополией, но исследования всегда должны были себя окупать. В действительности исследователи любят находить проблемы для исследования. Одно из преимуществ проведения научных исследований в большой компании — громадный диапазон встающих перед вами задач. Например, теоретики могут развивать теорию компиляции или алгоритмы для БИС, численные аналитики — изучать распределение заряда и тока в полупроводниках, а создатели программного обеспечения, конечно же, любят разрабатывать системы и писать программы для пользователей. Таким образом, исследования по информатике в Bell Laboratories всегда имели значительный практический выход, и ее сотрудники не боялись указов, велящих нам быть практиками.

Для некоторых из нас фактически главным камнем преткновения стала неспособность убедить других, что продукты наших исследований на самом деле могут принести пользу. Можно изобрести новое приложение, написать демонстрационную программу и внедрить ее в нашей собственной лаборатории. Многие такие начинания требуют дальнейшего развития и непрерывной поддержки, чтобы компания могла наилучшим образом использовать их. В прошлом они использовались исключительно внутри Bell System. Позднее появилась возможность разработки продукта для прямой продажи.

Например, несколько лет назад Майк Леск разработал автоматизированную телефонно-справочную систему [3]. Эта программа имела встроенный телефонный справочник Bell Laboratories и была подсоединена к синтезатору речи и анализатору

ру тона, подключенным к телефонной линии. Пользователь вызывал систему по телефону и набирал на клавиатуре телефона имя и код территории абонента; в ответ система произносила телефонный номер абонента и координаты его рабочего места (она не пыталась произносить имя). Несмотря на то что приходилось довольствоваться всего 12 кнопками (из-за чего, к примеру, «А», «В» и «С» склеивались вместе), система была достаточно точной: она давала примерно 5 процентов отказов. Эта программа имела большой успех в компании и широко использовалась. К несчастью, мы не смогли найти никого, кто бы взял ее под свою опеку, даже как сопровождаемую услугу внутри компании, хотя бы на общественных началах, и эта система слишком дорого нам обходилась и была поэтому в конце концов сдана в утиль. (Я выбрал этот пример не только потому, что он достаточно стар и никого сейчас не обидит, но и потому, что он своевременен: организация, издающая телефонную книгу компании, недавно запрашивала нас, нельзя ли восстановить данную систему.)

Конечно, не каждая идея заслуживает развития и поддержки. Как бы то ни было, мир меняется. За нашими идеями и советами охотятся гораздо более жадно, чем прежде. Этот рост влияния продолжался в течение нескольких лет, частично из-за успеха UNIX, а в последнее время — из-за драматического изменения структуры нашей компании.

Фирма AT&T предоставила самостоятельность своим операционным телефонным компаниям с начала 1984 г. Было высказано множество предположений по поводу того, как это отразится на фундаментальных исследованиях в Bell Laboratories. Типичен доклад в Science [2]. Одно из опасений, которое иногда высказывается, состоит в том, что фундаментальные исследования могут вообще зачахнуть, так как они будут приносить мало краткосрочных выгод для новой, меньшей AT&T. Официальная позиция компании — успокоение. Более того, научное руководство в Bell Laboratories, кажется, глубоко верит и убедительно аргументирует, что обязательство поддерживать фундаментальные исследования дано всерьез и надолго [1].

За фундаментальные исследования по физике, химии и математике в Bell Laboratories можно на самом деле не опасаться; тем не менее может возникнуть опасность, что они окажутся ненужными для целей компании, и надо быть готовыми доказывать обратное. Исследования по информатике отличаются от этих более традиционных дисциплин. С философской точки зрения отличие от физики состоит в том, что информатика не пытается открывать, объяснять или использовать окружающий мир, а вместо этого изучает свойства машин, созданных человеком. В этом она аналогична математике, и в самом деле,

«научная» часть информатики большей частью носит по своему духу математический характер. Но неотъемлемый аспект информатики — создание компьютерных программ: объектов, которые, хотя и неосвязаемы, являются предметами торгового обмена.

Самую большую опасность для успешных исследований по информатике сегодня может представлять ее *чрезмерная* практическая значимость. Признаки всемирного увлечения компьютерами — всюду: от статей по финансовым вопросам и даже передовиц газет до трудностей, которые испытывают наиболее престижные университеты в поиске и удержании дарований в области информатики. Лучшие профессора вместо обучения первоклассных студентов вступают во вновь образуемые компании и часто обнаруживают, что наиболее одаренные студенты, которых они обучали, уже опередили их. Информатика — в центре внимания, особенно те ее аспекты, которые, как, например, системы, языки и архитектура машин, могут иметь непосредственное коммерческое применение. Это внимание льстит, но оно может идти в ущерб качеству исследовательской работы.

По мере того как интенсивность исследований в какой-либо конкретной области возрастает, растет и побуждение сохранять их результаты в тайне. Это верно даже в университетах (хорошо известный пример — отчет Уотсона [12] о раскрытии структуры ДНК), хотя в академической науке есть сильное противодействие: без публикаций нет известности. В промышленной сфере защита информации, составляющей собственности, — естественный залог процветания. Исследователи понимают разумные ограничения на то, что и когда им публиковать, но многие из них будут раздражены и сбегут в другое место либо станут работать в менее деликатных областях, если им запретят подходящим образом обнародовать их открытия и изобретения. Научное руководство Bell Laboratories традиционно всегда поддерживало тщательный баланс между интересами компании и промышленным эквивалентом академической свободы. Вступление AT&T в сферу компьютерной промышленности испытывает на прочность этот баланс.

Другая опасность заключается в том, что того или иного рода коммерческое давление отвлечет внимание лучших умов от действительной новизны к эксплуатации текущих идей, от разведки новых областей к разработке старых жил. Это давление проявляется не только в утечке мозгов из науки в промышленность, но также и в консерватизме, который овладевает теми, кто получает хороший доход от вклада — интеллектуального или финансового — в данную разработку. Возможно, этот эффект объясняет, почему так мало интересных програм-

мных систем является продуктом больших компаний по производству компьютеров; эти компании замкнуты в своей оболочке. Даже IBM, которая поддерживает штат продуктивно работающих ученых с хорошей репутацией, в последние годы произвела немного такого, что вызвало хотя бы маленькую революцию в представлении людей о компьютерах. Кажется, что примеры реально используемых важных новых систем являются либо результатом самостоятельной инициативы (хороший пример — Visicalc), либо продуктов больших компаний типа Bell Laboratories, и особенно Xerox, которые были сильно связаны с компьютерами и могли позволить себе исследования по ним, но не считали их своим основным делом.

С другой стороны, в более мелких компаниях даже самая энергичная поддержка научных исследований сильно зависит от конъюнктуры рынка. Эту проблему отмечает The New York Times в статье, описывающей переход Алана Кая из Atari в Apple: «Господин Кай... сказал, что лаборатории Atari потеряли часть атмосферы новизны, которая некогда привлекла несколько выдающихся талантов в промышленность». «Когда в прошлом месяце я ушел оттуда, мне стало ясно, что они направят свои усилия на краткосрочные цели, — сказал он. — Я полагаю, что древо научных исследований должно время от времени удобряться кровью крохоборов» [9].

Частью потому, что они молоды и еще незрелы, а частью потому, что они — продукты интеллекта, искусство и наука программирования сокращают обычную для физики и техники цепочку от фундаментальных открытий через перспективные разработки к применению. Изобретатели новшества в программном обеспечении обычно встают перед необходимостью строить демонстрационные системы. Для больших систем и революционных идей требуется много времени: можно сказать, что система UNIX была написана в 70-е годы для того, чтобы выкристаллизовать лучшие идеи 60-х годов в области операционных систем, и стала банальностью в 80-е годы. Работы в Xerox PARC по персональным компьютерам, растровой графике и средам программирования [10] демонстрируют аналогичную прогрессию, начавшись и придя к осуществлению на несколько лет позже. Время и приверженность к долгосрочной ценности исследований требуются как со стороны исследователей, так и со стороны их руководства.

Bell Laboratories дала такие обязательства и даже больше: необыкновенное и уникально стимулирующее научно-исследовательское окружение для моих коллег и меня. На новом этапе, называемом в публикациях компании «новой конкурентной эрой», ее руководители и сотрудники хорошо сделают, если будут помнить, как и при каких обстоятельствах стала успеш-

ной система UNIX. Если мы сможем поддержать достаточную открытость для новых идей, достаточную свободу общения, достаточное терпение, чтобы позволить расцвести новому, то, возможно, будущий Кен Томпсон найдет малоиспользуемую машину CRAY/I и снарядит ее системой, такой же созидательной и такой же влиятельной, как UNIX.

#### ЛИТЕРАТУРА

1. Bell Labs. New order augurs well. *Nature* 305, 5933 (Sept. 29, 1983).
2. Bell Labs on the brink. *Science* 221 (Sept. 23, 1983).
3. Lesk M. E. User-activated BTL directory assistance. Bell Laboratories internal memorandum (1972).
4. Norman D. A. The truth about UNIX. *Datamation* 27, 12 (1981).
5. Organick E. I. *The Multics System*. MIT Press, Cambridge, MA, 1972.
6. Ritchie D. M. UNIX time-sharing system: A retrospective. *Bell Syst. Tech. J.* 57, 6 (1978), 1947—1969.
7. Ritchie D. M. The evolution of the UNIX time-sharing system. In *Language Design and Programming Methodology*, Jeffrey M. Tobias, ed., Springer-Verlag, New York (1980).
8. Ritchie D. M. and Thompson, K. The Unix time-sharing system. *Commun. ACM* 17, 7 (July 1974), 365—375.
9. Sanger D. E. Key Atari scientist switchers to Apple. *The New York Times* 133, 46, 033 (May 3, 1984).
10. Thacker C. P. et al. Alto, a personal computer, Xerox PARC Technical Report CSL—79—11.
11. Thompson K. Unix time-sharing system: UNIX implementation. *Bell Syst. Tech. J.* 57, 6 (1978), 1931—1946.
12. Watson J. D. *The Double Helix: A Personal Account of the Discovery of the Structure of DNA*. Atheneum Publishers, New York (1968).

1983

# Размышления о том, можно ли полагаться на доверие

*Кен Томпсон*

AT&T Bell Laboratories

(Кен Томпсон стал солауреатом премии Тьюринга 1983 г. за участие в разработке и реализации операционной системы UNIX. См. предисловие к лекции Д. М. Ритчи «Размышления об исследованиях в области программного обеспечения» на с. 195).

До какой степени можно полагаться на утверждение, что программа не содержит «троянских коней»? Возможно, более важно — полагаться на людей, написавших эту программу.

## ВВЕДЕНИЕ

Я благодарен ACM за эту награду, но не могу не чувствовать, что я обязан ею не только техническим достоинствам системы UNIX, но и тому, в какой удачный момент времени она появилась. Система UNIX завоевала популярность в связи с широким поворотом промышленности от больших центральных ЭВМ к автономным мини-машинам. Я подозреваю, что вместо меня здесь должен был бы быть Даниэл Бобров [1], если бы ему пришлось иметь дело с PDP-11, а не с PDP-10. Более того, UNIX в ее современном виде — результат труда многих людей.

Есть старая поговорка: «Танцуй с тем, кто тебя привел», согласно которой я должен говорить о UNIX. Я не участвовал в основном русле работ по UNIX в течение многих лет, однако продолжаю получать незаслуженную честь за труды других. Поэтому я не собираюсь говорить о UNIX, но хотел бы поблагодарить всех, кто внес в нее свой вклад.

Это наводит меня на мысль о Дэнисе Ритчи. Наше сотрудничество было образцом совершенства. За десять лет, которые мы проработали вместе, я могу вспомнить только один случай нескоординированной работы. Я тогда обнаружил, что мы оба написали одинаковую ассемблерную программу из 20 строк. Я сравнил наши тексты и был поражен, обнаружив, что они совпадают посимвольно. Результат нашей совместной работы был намного больше, чем вклад нас обоих по отдельности.



Я программист. Именно так я пишу в анкетах в графе «род занятий». Как программист, я пишу программы. Я хотел бы представить вам кратчайшую программу, которую я когда-либо написал. Я буду делать это в три этапа, а в конце попытаюсь собрать их в единое целое.

## ЭТАП I

В колледже, когда еще не было видеоигр, мы, бывало, развлекались упражнениями в программировании. Одним из любимых упражнений было написать кратчайшую самовоспроизводящуюся программу. Так как это упражнение далеко от реальности, обычным инструментом был Фортран. На самом деле Фортран был выбран по той же причине, по которой популярны скачки на стреноженных лошадях.

Более того, задача состоит в написании исходной программы, которая после компиляции и выполнения выдает в качестве вывода точную копию своего исходного текста. Если вам никогда не приходилось решать эту задачу, я настоятельно рекомендую попробовать сделать это самостоятельно. Удивительное открытие намного выше любой выгоды, которую вы можете извлечь, если вам скажут, как надо это делать. Требование насчет наименьшей длины было просто стимулом, чтобы прожить сноровку и определить победителя.

На рис. 1 показана самовоспроизводящаяся программа на языке программирования Си [3]. (Дотошный читатель заметит, что эта программа не является в точном смысле самовоспроизводящейся, однако выдает в качестве результата самовоспроизводящуюся программу.) Этот текст слишком длинный, чтобы завоевать приз, но он демонстрирует прием решения и имеет два важных свойства, которые потребуются для завершения моего рассказа: 1) эта программа может быть легко порождена другой программой; 2) эта программа может содержать произвольное количество добавочного текста, который будет воспроизводиться наряду с основным алгоритмом. В приведенном примере воспроизводится даже комментарий.

## ЭТАП II

Компилятор языка Си написан на Си. То, о чем я собираюсь рассказать, представляет собой одну из разновидностей проблемы курицы и яйца, которая возникает, когда компиляторы пишутся на своем собственном языке. В данном случае я буду использовать конкретный пример из компилятора Си.

Си позволяет задавать строковую константу в виде инициализированного массива символов (литер). Для обозначения

```

s[] = {
    '\n',
    '0',
    '\n',
    '|',
    '\n',
    '\n',
    '/',
    '\n',
    (213
    0
};

/*
 *
 *
 */

( )
{
    i;

    ("char\ts[ ] = {\n}");
    (i=0; s[i]; i++)
        ("\t%d, \n", s[i]);
    ("%s", s);
}

=
==
!=
++
'x'
"xxx"
%d
%s
\
\n

```

Рис. 1.

некоторых символов, не имеющих графического представления, в строке могут использоваться управляющие последовательности. Например,

«Здравствуй, мир \n»

является строкой символов, в которой «\n» представляет переход на новую строку.

На рис. 2.1 представлен фрагмент программы компилятора Си, в котором интерпретируются управляющие последовательности символов. Это удивительный фрагмент. Он «знает» полностью переносимым образом, какой код символа компилируется для новой строки в любой кодировке символов. Этот элемент знания позволяет ему перекомпилировать себя, увековечив таким образом знание.

Предположим, мы хотим изменить компилятор Си, чтобы включить управляющую последовательность «\v» для представления символа вертикальной табуляции. Расширение фрагмента 2.1 очевидно и представлено на рис. 2.2. Затем мы заново компилируем компилятор Си, но получаем сообщение об ошибке<sup>1)</sup>. Ясно, что, поскольку компилятор Си в готовом виде ничего не знает о «\v», исходная программа не является правильной программой на языке Си. Заглянув в таблицу кодировки символов ASC II, мы находим, что код вертикальной табуляции — десятичное 11. Мы изменяем нашу исходную программу, как показано на рис. 2.3. Теперь старый компилятор компилирует новый исходный текст без ошибок. Мы установ-

```
...
c = next( );
if(c != '\\')
    return(c);
c = next( );
if(c == '\\')
    return('\\');
if(c == 'n')
    return('\n');
...
```

Рис. 2.1.

```
...
c = next( );
if(c != '\\')
    return(c);
c = next( );
if(c == '\\')
    return('\\');
if(c == 'n')
    return('\n');
if(c == 'v')
    return('\v');
...
```

Рис. 2.2.

```
...
c = next( );
if(c != '\\')
    return(c);
c = next( );
if(c == '\\')
    return('\\');
if(c == 'n')
    return('\n');
if(c == 'v')
    return('\n');
...
```

Рис. 2.3.

<sup>1)</sup> На самом деле сообщения об ошибке не будет, но будет сгенерирован не тот код. Согласно правилам Си (см., например, [2]), «\v» будет интерпретироваться как символ «v», если эта управляющая последовательность неизвестна компилятору. — *Прим. перев.*

ливаем полученную готовую программу как новую официальную версию компилятора Си и теперь можем написать мобильную версию, как это сделано на рис. 2.2.

Это глубокая идея. В моем понимании она близка к понятию «обучающейся» программы. Вы просто сообщаете новое знание один раз, а затем можете использовать определение, ссылающееся само на себя.

### ЭТАП III

Снова компилятор Си. На рис. 3.1 показан фрагмент компилятора Си, занимающий высокий уровень в его структуре. Подпрограмма «compile» вызывается для компиляции каждой следующей строки исходного текста. На рис. 3.2 показана модификация компилятора, которая преднамеренно неправильно компилирует исходную программу, когда в ней встречается заданный образец. Когда такое делается не преднамеренно, это называется ошибкой компилятора, или «жучком». Так как это сделано преднамеренно, то говорят, что мы имеем дело с «троянским конем».

Действительный «жучок», который я встроил бы в компилятор, распознавал бы исходный текст команды login (вход в систему UNIX). Взамен подставлялся бы такой исходный текст, чтобы команда login принимала как правильный заранее известный пароль (в зашифрованном<sup>1)</sup> или обычном виде). Таким образом, если бы такой компилятор был установлен в готовом (двоичном) виде и был использован для компиляции команды login, я мог бы войти в такую систему, как и любой другой пользователь.

```
compile(s)
char *s;
{
    ...
}
```

Рис. 3.1.

```
compile(s)
char *s;
{
    if(match(s, "pattern")) {
        compile("bug");
        return;
    }
    ...
}
```

Рис. 3.2.

<sup>1)</sup> Все пароли в UNIX хранятся в зашифрованном виде в обычных текстовых файлах, в принципе доступных на чтение всем пользователям. — *Прим. перев.*

```
compile(s)
char *s;
{
    if (match(s, «образец 1»)) {
        compile («жучок 1»);
        return;
    }
    if (match(s, «образец 2»)) {
        compile («жучок 2»);
        return;
    }
    ...
}
```

Рис. 3.3.

Такая наглая вставка в программе не долго оставалась бы незамеченной. Даже при случайном внимательном просмотре исходного текста компилятора Си она вызвала бы подозрение.

Заключительный шаг представлен на рис. 3.3. Он заключается в простом добавлении еще одного «троянского коня» к уже существующему. Второй образец настроен на сам компилятор Си. Заменяющим текстом служит самовоспроизводящаяся программа (см. этап I), которая вставляет обоих «троянских коней» в компилятор. Потребуется также фаза обучения, как в примере из этапа II. Сначала мы компилируем модифицированный исходный текст нормальным компилятором Си и получаем готовую программу с «жучком». Затем устанавливаем эту готовую программу как официальный компилятор Си. Теперь можно удалить жучки из исходного текста компилятора, а новый компилятор будет воспроизводить жучки при каждой перекompilации. Команда `login`, естественно, будет оставаться с «жучком» без всякого следа в каких-либо исходных текстах.

## МОРАЛЬ

Мораль ясна. Нельзя доверять программе, которую вы не написали полностью сами (особенно если эта программа пришла из компании, нанимающей таких людей, как я). Сколько бы вы не исследовали и не верифицировали исходный текст — это не защитит вас от троянской программы. Для демонстрации атаки такого рода я выбрал компилятор Си. Я мог бы выбрать любую программу, обрабатывающую другие программы — ассемблер, загрузчик или даже микропрограмму, зашитую в аппаратуру. Чем ниже уровень программы, тем труднее и труднее обнаруживать подобные «жучки». Мастерски встроенный «жучок» в микропрограмме будет почти невозможно обнаружить.

Теперь, после того как я попытался убедить вас, что мне нельзя доверять, я желаю прочесть мораль. Я хотел бы покритиковать прессу за ее отношение к «хэккерам»: банда 414, банда Далтона и т. п. Действия, совершаемые этими «детками», в лучшем случае — вандализм, а в худшем — нарушение прав владельцев и воровство. Только неадекватность уголовного кодекса спасает хэккеров от очень сурового наказания. Компании, которым угрожает такого рода деятельность (а большинство крупных компаний подвергается очень большому риску), усиленно настаивают на изменении уголовного кодекса. Несанкционированный доступ к компьютерным системам уже сейчас является серьезным преступлением в нескольких штатах, и в настоящий момент этот вопрос рассматривается законодательными органами во многих других штатах, а также в Конгрессе.

Назревает взрывоопасная ситуация. С одной стороны, пресса, кино и телевидение делают героев из хулиганов, называя их озорными детками. С другой стороны, действия, совершаемые этими детками, скоро будут наказываться годами лишения свободы.

Я видел этих деток, дававших показания в Конгрессе. Ясно, что они абсолютно не подозревают о серьезности своих действий. Очевидно, мы имеем здесь пробел в культуре. Проникновение в компьютерную систему в общественном мнении должно быть таким же неблагоприятным поступком, как проникновение в дом соседа. Не должно иметь значения, что дверь у соседа открыта. Пресса должна учить, что использование компьютера не по назначению ни капли не смешнее, чем пьянство за рулем.

## БЛАГОДАРНОСТИ

Я впервые прочитал о возможности такого троянского коня в документе военно-воздушных сил [4], в котором критикуются средства защиты от несанкционированного доступа ранней реализации системы Multics. Я не могу найти более точной ссылки на этот документ. Я буду благодарен тому, кто может найти эту ссылку и даст мне об этом знать.

## ЛИТЕРАТУРА

1. Bobrow D. G., Burchfiel, J. D., Murphy, D. L., and Tomlinson, R. S. TENEX, a paged time-sharing system for the PDP-10. Commun. ACM 15, 3 (Mar. 1972), 135—143.
2. Kernighan B. W., and Ritchie, D. M. The C Programming Language. Prentice-Hall, Englewood Cliffs, N. J., 1978. [Имеется перевод: Керниган Б., Ритчи Д. Язык программирования Си. Москва. Финансы и статистика, 1985.]
3. Ritchie D. M., and Thompson K. The UNIX time-sharing system. Commun. ACM 17, 7 (July 1974), 365—375.
4. Неизвестный документ Военно-воздушных сил.

1984

# От разработки языка программирования к созданию компьютера

*Никлаус Вирт*

Никлаус Вирт из Швейцарского государственного технологического института (ETH) был награжден премией Тьюринга в 1984 г. на ежегодной конференции Ассоциации, состоявшейся в Сан-Франциско в октябре. Это награждение стало признанием его выдающихся заслуг в области разработки ряда новейших языков программирования: Эйлер, Алгор-W, Модула и Паскаль. В частности, язык Паскаль приобрел особое значение в сфере образования и заложил базу для будущих исследований в области языков программирования, систем и архитектуры. Отличительными чертами разработанных Виртом языков являются простота, экономичность и высокое качество проработки деталей, что в конечном итоге приводит к языкам, система обозначений которых скорее является естественным продолжением алгоритмического образа мышления, а не чуждого ему формализма.

Талант Вирта как разработчика языков программирования дополняется писательским даром. В апрельском номере 1971 г. журнала Communications of the ACM Вирт поместил основополагающую работу по структурному программированию («Разработка программы методом поэтапного усовершенствования»), которая рекомендовала нисходящее, идущее сверху вниз, построение программы (т. е. последовательное уточнение кусков программы, пока она не окажется полностью проработанной). Полученный в результате элегантный и мощный метод описания представляет интерес и сегодня, даже несмотря на то, что восторги относительно структурного программирования поубавились. Две более поздние статьи «О дисциплине программирования в реальном времени» и «Что мы можем сделать с необязательным разнообразием обозначений», опубликованные в том же журнале соответственно в августе и ноябре 1974 г., говорят о последовательном и неотступном поиске Виртом адекватного языкового формализма.

Премия Тьюринга, являющаяся высшим признанием со стороны Ассоциации по информатике вклада в дело вычислительного сообщества, учреждена в честь английского математика

Алана М. Тьюринга, создавшего прототип компьютера — машину Тьюринга — и помогшего раскрыть шифры Германии во время второй мировой войны.

Вирт получил степень доктора философии в Калифорнийском университете в Беркли в 1963 г. и был приглашенным профессором в Станфордском университете до 1967 г. С 1968 г. он является профессором Швейцарского государственного политехнического института в Цюрихе; с 1982 по 1984 г. он возглавлял в этом же институте факультет информатики. Последняя работа Вирта связана с конструированием и разработкой персонального компьютера Лилит в сочетании с использованием языка Модула-2. В своей лекции Вирт кратко излагает историю своих основных проектов, описывает их результаты и формулирует принципы, которые направляли его работу.

Путь, пройденный Виртом в поиске приемлемого формализма системного программирования, начиная с NELIAC, через Алгол-60 к языкам Эйлер и Алгол-W, Паскаль, Модула-2 и, в конечном итоге, до Лилит, полон впечатляющих открытий и удивительных результатов.

Получить премию Тьюринга доставляет огромное удовольствие, а признание работы, которая проводилась в течение стольких лет, одновременно радует и вдохновляет. Я хочу поблагодарить АСМ за присуждение мне этой престижной награды. Особое значение имеет то, что я получаю ее в Сан-Франциско, где началась моя профессиональная карьера.

Вскоре после того, как я узнал о присуждении премии, мое чувство радости было несколько омрачено необходимостью представить Тьюринговскую лекцию. У человека, являющегося прежде всего инженером, а не оратором или проповедником, эта обязанность вызывает заметное беспокойство. Главный среди возникающих вопросов — чего в первую очередь ждут люди от такой лекции? Одни хотят проникнуть в суть чьей-либо работы или услышать оценку ее важности или ожидаемого воздействия. Другие захотят услышать, как возникли идеи, лежащие в ее основе. Третьи ждут авторитетного мнения относительно будущих тенденций, событий и товарных продуктов. Кто-то надеется услышать откровенную оценку современного положения дел: восторги, вызванные впечатляющим прогрессом нашей науки, или же сетования на негативные побочные эффекты и преувеличения.

В нерешительности я просмотрел некоторые из предыдущих Тьюринговских лекций и пришел к выводу, что лаконичное сообщение об истории какой-либо работы будет наиболее приемлемым. Чтобы это не было всего лишь развлечением, я попытался подытожить все то, чему, как я считаю, мне удалось научиться за прошедшие годы. Откровенно говоря, этот выбор вполне устраивает меня, поскольку я ни в коей мере не претен-



дую на то, что знаю о будущем больше, чем большинство присутствующих, и к тому же совершенно не хочу оказаться впоследствии неправым. В то же время искусство читать проповеди о нынешних достижениях и прогрешениях отнюдь не является моим коньком. Это не означает, что я равнодушно наблюдаю происходящее в области информатики, в частности, шумную борьбу с коммерциализацией.

Конечно, когда в 1960 г. я вступил в область информатики, ей не уделялось столь большого внимания ни в коммерческой рекламе, ни в академических учебных планах. Во время моего обучения в Швейцарском государственном технологическом институте единственное упоминание о компьютерах, которое я слышал, прозвучало в факультативном курсе, читавшемся Амброзом Спайзером, ставшим позднее президентом Международной федерации по обработке информации (IFIP). Разработанный им компьютер ERMETH был малодоступен обычным студентам, и поэтому мое посвящение в информатику оказалось отложенным до того момента, как я прослушал курс численного анализа в Лавальском университете в Канаде. Но, увы, компьютер «Алвак-IIIЕ» большую часть времени был неисправен, и упражнения по программированию так и остались на бумаге в виде непроверенных последовательностей шестнадцатеричных кодов.

Моя следующая попытка оказалась несколько более удачной: в Беркли я столкнулся с любимцем Гарри Хаски — компьютером «Бендиск G-15». Хотя этот компьютер позволил ощутить что-то вроде успеха, поскольку на нем удавалось получать результаты, суть искусства программирования представляла как удачное размещение команд на барабане. Если вы пренебрегали этим искусством, то ваши программы вполне могли работать раз в 100 медленнее. Однако его учебная ценность была налицо: вы не могли позволить себе игнорировать самую незначительную из мелких деталей. Такого простого способа компенсировать недостатки программы, как приобретение дополнительной памяти, не существовало. Теперь вспоминается как самая притягательная черта, что каждая деталь машины была видна и ее назначение можно было понять. Ничего не было скрыто в сложной электронной схеме, кристаллах кремния или в загадочной операционной системе.

С другой стороны было очевидно, что программирование будущих компьютеров должно быть более эффективным. Поэтому я отказался от идеи изучить вначале, как проектировать аппаратную часть, в пользу того, чтобы научиться более элегантно ее использовать. Мне повезло, что я присоединился к группе, участвовавшей в разработке — или, скорее, в доработке — компилятора и в его использовании на IBM-704. Этот язык

был назван NELIAC — диалект языка Алгол-58. Преимущества такого «языка» быстро стали очевидны, а задача автоматической трансляции программ в машинный код ставила трудные вопросы. Это была как раз та ситуация, к которой стремятся при подготовке докторской диссертации. Компилятор, который тоже был написан на языке NELIAC, представлял собой нечто чрезвычайно запутанное. Казалось, что он состоит на 1% из науки, а на 99% — из колдовства. Этот перекосяк и предстояло ликвидировать. Ясно, что программы должны строиться в соответствии с теми же принципами, что и электронные схемы, четко разбивающиеся на блоки, границы которых пересекают всего несколько проводов. Только поняв, как действует каждая часть в отдельности, можно надеяться в конце концов понять, как действует целое.

Благодаря официальному Сообщению об Алголе-60 эта попытка получила сильный начальный импульс. Алгол-60 стал четко определенным языком, а его синтаксис даже определялся с помощью строгого формализма. Урок заключался в том, что ясное определение является необходимым, но недостаточным условием надежной и эффективной реализации. Контакт с Адрианом ван Вейнгаарденом — одним из разработчиков Алгола — позволил яснее выявить центральную тему: можно ли в еще большей степени сконцентрировать и выкристаллизовать принципы Алгола?

Вот так начались мои приключения в области языков программирования, напоминавшие путешествие с мачете сквозь джунгли особенностей и возможностей языка. Первый эксперимент привел к диссертации и языку Эйлер. Результат оказался академически элегантным, но имел весьма малую практическую ценность — он был почти антитезой более поздним языкам с типами данных и языкам структурного программирования. Но он действительно заложил фундамент систематической разработки компиляторов, позволявших без потери ясности — по крайней мере такова была надежда — расширять их, чтобы включать новые возможности.

Язык Эйлер привлек внимание Рабочей группы Международной федерации по обработке информации, участвовавшей в составлении планов относительно будущего Алгола. Язык Алгол-60, созданный специалистами в области численных методов для своих целей, обладал систематической структурой и четким определением, что было оценено людьми с математической подготовкой. Однако ему не хватало компиляторов и поддержки со стороны промышленности. Чтобы завоевать эту поддержку, необходимо было расширить сферу его применения. Рабочая группа взяла на себя задачу предложить преемника этого языка и вскоре распалась на два лагеря. Один состоял

из честолюбцев, желавших заложить еще один краеугольный камень в разработку языков, а по другую сторону находились те, кто чувствовал, что время торопит и что должным образом расширенный Алгол-60 может оказаться продуктивным. Я находился в этом втором лагере и выдвинул предложение, которое не нашло у группы достаточной поддержки. Впоследствии это предложение было улучшено с помощью Тони Хоара (члена той же группы) и реализовано на первой IBM-360 Станфордского университета. Позже этот язык стал известен под названием Алгол-W и использовался в нескольких университетах в учебных целях.

Стоит упомянуть и небольшую паузу в этой масштабной разработке. Новая IBM-360 предлагала только ассемблер и, естественно, Фортран. Ни то, ни другое ни у меня, ни у моих студентов не вызывало особой симпатии как инструмент для создания компилятора. Поэтому я набрался храбрости ввести еще один новый язык, на котором мог бы быть описан компилятор Алгола: некий компромисс между Алголом и возможностями, предоставляемыми ассемблером, он должен был служить машинным языком, структура операторов и описание команд которого напоминают Алгол. Примечательно, что описание языка было подготовлено за пару недель; я написал кросскомпилятор для компьютера «Барроуз В-5000» за четыре месяца, а один прилежный студент примерно за такой же отрезок времени переписал его для IBM-360. Эта подготовительная пауза помогла существенно ускорить работу с Алголом. Хотя первоначально считалось, что новый язык будет использоваться только для наших насущных нужд и потом будет забыт, он быстро приобрел собственное значение. Язык PL360 стал эффективным инструментом во многих областях и вдохновил на осуществление аналогичных разработок для других машин.

По иронии успех PL360 вместе с тем являлся указанием на неудачу языка Алгол-W. Диапазон применений Алгола был расширен, но в качестве инструмента для системного программирования он все еще обладал явными недостатками. Трудно, как мы убедились, удовлетворить сразу многим требованиям с помощью одного языка, да и сама цель оказалась под вопросом. Язык PL/I, созданный приблизительно в это же время, предоставил дополнительные доказательства в пользу этой точки зрения. Идея, использованная в швейцарском армейском ноже, обладает рядом достоинств, но если с ней переборщить, то этот нож превратится в обузу. К тому же размер компилятора для языка Алгол-W вышел за рамки, в которых можно чувствовать себя уютно благодаря возможности понять всю программу в целом. Стремление к еще более четкому и в то же время адекватному формализму для системного программирова-

ния оказалось нереализованным. Для системного программирования необходим эффективный компилятор, генерирующий эффективный код, который работает без фиксированного, «запакованного» и большого пакета программ так называемого времени прогона. Достичь этого не удалось ни на Алголе, ни на PL/I, поскольку оба языка были сложными, а компьютеры, для которых они писались, для них не подходили.

Осенью 1967 г. я вернулся в Швейцарию. Еще через год я смог создать группу из трёх помощников для внедрения языка, ставшего позднее известным как Паскаль. Избавленный от необходимости получить единогласную поддержку Комитета, я смог сосредоточить основное внимание на том, чтобы включить те характеристики, которые считал существенными, и выбросить те, которые, на мой взгляд, не окупали усилий по их реализации. В некоторых ситуациях жестко ограниченное число сотрудников является преимуществом.

Утверждалось, что Паскаль был разработан в качестве языка для обучения. Хотя это утверждение справедливо, но его использование при обучении не являлось единственной целью. На самом деле я не верю в успешность применения во время обучения таких инструментов и формализмов, которые нельзя использовать при решении каких-то практических задач. По сегодняшним меркам Паскаль обладал явными недостатками при программировании больших систем, но 15 лет назад он представлял собой разумный компромисс между тем, что было желательно, и тем, что было эффективно. В Швейцарском государственном политехническом институте мы начали использовать Паскаль на занятиях по программированию в 1972 г., встретив при этом серьезное сопротивление. Этот шаг оказался успешным, поскольку он позволил преподавателям уделять больше внимания конструкциям и концепциям, а не отдельным особенностям и характеристикам, т. е. принципам, а не технике.

Наш первый компилятор Паскаля был реализован на семействе компьютеров CDC-6000 и написан на Паскале. Никакого PL6000 не потребовалось, и я рассматривал это как существенный шаг вперед. Тем не менее генерируемый код был, без сомнения, хуже кода, генерируемого компиляторами Фортрана для соответствующих программ. Скорость является важным и легко поддающимся количественной оценке критерием, и мы считаем, что обоснованность концепции языка высокого уровня может быть воспринята промышленностью только в том случае, если потери эффективности скомпилированных кодов будут устранены полностью или по крайней мере сведены к минимуму. С учетом этого следующим шагом, причем предпринятым одним человеком, стала попытка создать высококачественный компилятор. Эта цель была достигнута в 1974 г. Урсом

Амманном, и компилятор впоследствии получил широкое распространение и продолжает использоваться сегодня многими университетами и промышленностью. И все же цена оставалась высокой; усилия по генерации хорошего (т. е. даже не оптимального) кода пропорциональны несоответствию языка машине, а CDC-6000, конечно, не была спроектирована с расчетом на языки высокого уровня.

И опять главный выигрыш проявился там, где мы его меньше всего ожидали. После того, как стало известно о существовании Паскаля, несколько человек попросили нас помочь в реализации Паскаля на различных машинах, подчеркивая, что они намерены использовать его для обучения, и быстрое действие не имеет первостепенного значения. После этого мы решили создать версию компилятора, которая генерировала бы код для машины нашей собственной конструкции. Позднее этот код стал известен как Р-код. Создать эту версию компилятора было очень легко, поскольку новый компилятор создавался в качестве важного эксперимента в области структурного программирования путем последовательного уточнения и потому первые несколько шагов уточнений могут приниматься без изменений. Паскаль-Р оказался исключительно удачным для распространения языка среди большого числа пользователей. Если бы у нас хватило мудрости предвидеть масштабы такого развития событий, то мы приложили бы больше усилий и тщательности при разработке и документировании Р-кода. А тогда это представляло собой побочную деятельность, осуществляющуюся только для того, чтобы удовлетворить запросы одним сосредоточенным усилием. Это показывает, что даже с наилучшими намерениями можно выбрать неверные цели.

Но подлинно широкое признание Паскаль получил только после того, как Кен Боулес в Сан-Диего обнаружил, что Р-система с успехом может быть реализована на новых микрокомпьютерах. Его усилия по разработке подходящей среды с интегрированным компилятором, формирователем файла, редактором и отладчиком привели к прорыву: Паскаль стал доступен тысячам пользователей новых компьютеров. Эти пользователи не были обременены ранее приобретёнными привычками, и их не душила необходимость сохранять совместимость со старым программным обеспечением.

В это же время я закончил работу над Паскалем и решил заняться изучением нового заманчивого предмета — мультипрограммирования, для которого Хоар уже заложил солидный фундамент, а Бринк Хансен с помощью своего *Параллельного Паскаля* наметил путь. Попытка задать конкретные правила дисциплины мультипрограммирования быстро заставила меня сформулировать их в терминах небольшого набора средств

программирования. Чтобы подвергнуть эти правила настоящей проверке, я встроил их во фрагментированный язык, название которого отражало модульный принцип организации комплексов программ — мою основную цель. Позднее эта модульность оказалась основным достоинством данного языка: она позволила придать абстрактной концепции сокрытия информации конкретную форму и воплотила метод, одинаково важный как для обычного программирования, так и для мультипрограммирования. Кроме того, язык *Модула* также содержал средства для описания параллельных процессов и их синхронизации.

К 1976 г. мне уже надоели и языки программирования, и угнетающая обязанность писать хорошие компиляторы для существующих компьютеров, разработанных для устаревшего «ручного» кодирования. К счастью, мне представилась возможность провести годичный отпуск, предназначенный для научной работы, в исследовательской лаборатории корпорации «Ксерокс» в Пало-Альто, где не только родилась, но и нашла свое практическое воплощение идея мощных персональных рабочих станций. Вместо того, чтобы делить с многочисленными пользователями один большой компьютер и сражаться за свою долю, используя кабель с 3-килогерцевой полосой, я теперь через 15-мегагерцевый канал пользовался своим собственным компьютером, находившимся под моим рабочим столом. Последствия увеличения в 5000 раз предвидеть невозможно, слишком оно велико. Самым приятным было то, что после 16 лет работы на компьютеры теперь, похоже, компьютер работал на меня. Впервые я вместо того, чтобы намечать планы создания новых языков компиляторов и программ, которыми будут пользоваться другие, обрабатывал свою ежедневную почту и готовил доклад с помощью компьютера. Другим открытием стало то, что на такой рабочей станции можно было реализовать компилятор для языка *Меза* (MESA), сложность которого значительно превышала сложность компилятора для Паскаля. Эти новые условия работы на столько порядков превосходили все то, с чем я сталкивался дома, что я решил попытаться создать такие условия и там.

В конечном итоге я решил углубиться в разработку аппаратной части. Это решение подкреплялось моей давней неприязнью к существующим архитектурам компьютеров, которые отравляют жизнь разработчикам компиляторов, склонным к систематической простоте. Идея целиком разработать и построить компьютерную систему, состоящую из аппаратной части, микрокода, компилятора, операционной системы и программных средств, быстро обрела в моем воображении конкретные очертания конструкции, которая была бы свободна от любых ограничений вроде совместимости с PDP-11 или IBM-360, или

Фортраном, Паскалем, Юниксом и какими бы там еще ни было сиюминутными увлечениями или стандартами Комитета.

Однако для успешной реализации технического проекта одного чувства освобождения недостаточно. Упорная работа, уверенность, тонкое чувство того, что является важным, а что — эфемерно, а также некоторое везение необходимы. Первой удачей стал телефонный звонок одного разработчика аппаратной части, интересовавшегося возможностью приехать в наш университет для обучения в области разработки программного обеспечения и получения степени доктора философии. Почему бы не поучить его разработке программного обеспечения, чтобы он поучил нас, как разрабатывать аппаратное обеспечение? Вскоре мы превратились в одну команду, а Ричард Охрэн так заинтересовался новым проектом, что практически полностью забыл как о программном обеспечении, так и о докторской степени. Это не слишком сильно встревожило меня, поскольку я был достаточно занят конструированием частей аппаратного обеспечения, спецификациями микро- и макропрограмм, программированием интерпретатора макропрограмм, планированием полной системы программного обеспечения и, в частности, программированием редактора текста и графического редактора, которые оба использовали новый графический дисплей с высоким разрешением и маленькое чудо под названием «мышь» в качестве координатно-указательного устройства. Этот опыт написания сильно интерактивных сервисных программ потребовал изучения и использования методов, совершенно чуждых обычному проектированию компиляторов и операционных систем.

В целом проект был столь разносторонним и сложным, что приниматься за него казалось безответственным, особенно учитывая, что число наших помощников, лишь часть своего рабочего времени отводивших этой разработке, было невелико — около 7 человек. Основная угроза заключалась в том, что продолжительность этой работы могла оказаться слишком большой, чтобы энтузиазм сохранился у нас двоих и позволил остальным, еще не испытывавшим на себе огромных возможностей рабочей станции, стать такими же энтузиастами. Чтобы проект не превысил разумных размеров, я остановился на трех догмах: он предназначен для *однопроцессорного* компьютера, используемого *одним пользователем* и программируемого на *одном языке*. Заметим, что эти краеугольные положения были диаметрально противоположны существовавшим тенденциям, направленным на исследования по мультипроцессорным конфигурациям, операционным системам с разделением времени для большого числа пользователей и для такого числа языков, какое удастся найти.

Ограничившись одним языком, я столкнулся с трудным выбором, который мог иметь самые серьёзные последствия, а именно с выбором языка. Из существующих языков ни один не выглядел привлекательным. Они не могли удовлетворить всем требованиям, и к тому же ни в коей мере не устраивали разработчика компилятора, который знает, что задание должно быть выполнено в разумные сроки. В частности, язык должен был удовлетворять всем нашим положениям в отношении структурного программирования, основанным на десятилетнем опыте работы с Паскалем, и быть пригодным для тех задач, которые прежде решались исключительно при кодировании на ассемблере. Короче говоря, было решено создать потомка сразу двух языков: как уже апробированного Паскаля, так и экспериментального языка Модуля; им стал язык *Модуля-2*. Модульность — решающее свойство, позволяющее сочетать противоречивые требования: обеспечение надежности абстракции высокого уровня через проверку на избыточность и наличие средств низкого уровня, обеспечивающих доступ к индивидуальным конструктивным особенностям конкретного компьютера. Это позволяет программисту обособить использование средств низкого уровня в несколько небольших частей системы, защищаясь таким образом от непредвиденных осложнений.

Проект *Лилит* доказал, что разработка одноязыковой системы не только возможна, но и обладает рядом преимуществ. Буквально всё, начиная с драйверов для устройств и кончая графическим редактором и редактором текста, пишется на одном и том же языке. Никак не отличается подход к модулям, относящимся к операционной системе, и к модулям программы пользователя. Фактически это различие почти пропадает, и вместе с ним мы избавляемся от неизменного громоздкого резидентного блока программы, без которого всякий рад сбиться, но все вынуждены его использовать. Кроме того, проект *Лилит* доказал преимущества хорошей сочетаемости программного и аппаратного обеспечения. Эти преимущества можно оценить в терминах быстродействия: сравнение времен выполнения программ, написанных на Модуле, показало, что система *Лилит* часто предпочтительней, чем система VAX-750, сложность и стоимость которой многократно превышает сложность и стоимость проекта *Лилит*. Эти преимущества можно также оценить в терминах требуемого объема памяти: машинные коды программ, написанных на Модуле для *Лилит*, в 2—3 раза короче, чем для PDP-11, VAX или 68000, и в 1,5—2 раза короче, чем для NS-32000. Кроме того, части компилятора, генерирующие код, для этих микропроцессоров значительно запутаннее, чем аналогичные элементы системы *Лилит*, из-за неудачного набора команд. Этот фактор длины кода надо умно-



жить на низкий коэффициент плотности, который омрачает сильно разрекламированную пригодность языка высокого уровня для современных микропроцессоров и показывает, что эти претензии несколько преувеличены. Перспектива, что такие устройства будут выпускаться миллионными сериями, весьма удручает, поскольку благодаря одному их количеству они станут стандартной элементной базой. К сожалению, технология полупроводников развивалась столь быстро, что достижения в области компьютерной архитектуры оказались в тени и кажутся не столь важными. Конкуренция заставляет производителей «замораживать» новые разработки в виде серийно выпускаемых микросхем задолго до того, как они доказали свою эффективность. И в то время как громоздкое программное обеспечение может быть в худшем случае модифицировано, а в лучшем случае заменено, сегодня вся сложность спустилась непосредственно на уровень микросхем. Маловероятно, что мы лучше справляемся с вопросами сложности на уровне аппаратного обеспечения, чем на уровне программного.

И среди программистов, и среди инженеров-электронщиков найдется немало людей, для которых сложность есть и будет сильным притягательным моментом. Действительно, мы живем в сложном мире и стараемся решать сложные по своей сути проблемы, которые часто для своего решения требуют сложных устройств. Однако это не значит, что мы не должны стремиться найти *элегантные* решения, убеждающие своей ясностью и эффективностью. Простые элегантные решения более эффективны, но найти их *труднее*, чем сложные, и для этого требуется больше времени. Слишком часто мы считаем, что мы не можем себе позволить таких затрат.

Прежде чем закончить, я попытаюсь выделить некоторые общие характеристики упомянутых мною проектов. Очень важный метод, который редко используется столь же эффективно, как при вычислениях, это *раскрутка*. Мы использовали его почти в каждом проекте. При разработке любой системы, будь то язык программирования, компилятор или компьютер, я проектирую ее таким образом, чтобы она была полезной уже непосредственно на следующем этапе: PL360 был разработан для реализации Алгола-W, Паскаль — для реализации Паскаля, Модуль-2 для реализации программного обеспечения целой рабочей станции, Лилит — для обеспечения подходящей среды для всех наших будущих работ, начиная от написания программ и до документирования и разработки схемы, от подготовки отчета до проектирования шрифтов. С помощью раскрутки можно извлечь максимальную выгоду из затраченных усилий, но и сильнее всего пострадать от совершенных ошибок.

Все это вынуждает на *раннем этапе различать, что существенно, а что второстепенно*. Я всегда пытался выделить существенные моменты, которые дадут несомненные преимущества, и сосредоточиться на них. Например, включение в язык программирования четкой и согласованной схемы объявления типа данных я считал существенным, в то время как все мелочи, относящиеся к разнообразию циклов, или вопрос о том, должен ли компилятор различать прописные и строчные буквы, являлись для меня второстепенными. При проектировании компьютера я считал решающим выбор режимов адресации и обеспечения полными и согласованными наборами арифметических операций (со знаками или без знака), включающих прерывания по переполнению, а детали механизма приоритета прерываний при многоканальной системе — второстепенными. Еще важнее гарантия того, что решение второстепенных вопросов никогда не нарушит систематическое структурированное проектирование центральных устройств. Вместо этого второстепенные моменты должны подгоняться под существующую хорошо структурированную основу.

Иногда трудно устоять против настойчивых требований пользователей включить все типы средств, которые «хорошо было бы иметь». Опасность состоит в том, что стремление угодить чьему-то желанию нарушит согласованность всего проекта. Я всегда пытался сбалансировать выгоду и затраты. Например, размышляя, не включить ли некую языковую конструкцию или специальную обработку в компиляторе какой-то достаточно часто используемой конструкции, следует оценить выгоды и дополнительные расходы при реализации, поскольку одно только ее наличие сделает систему более громоздкой. Разработчики языка часто недооценивают этот момент. Я с готовностью допускаю, что временами было бы хорошо иметь некоторые возможности языка Ада, которые не имеют аналогов в Модуле-2, но в то же время я спрашиваю, стоят ли они таких затрат. Цена значительна. Во-первых, хотя создание обоих языков началось в 1977 г., компиляторы языка Ада начали появляться только сейчас, в то время как Модулой мы уже пользуемся с 1979 г. Во-вторых, ходят слухи, что компиляторы Ады — это гигантские программы, состоящие из нескольких сотен тысяч строк команд, в то время как наш новейший компилятор Модулы измеряется лишь 5 тысячами строк. Признаюсь по секрету, что этот компилятор Модулы уже находится на пределе того уровня сложности, который еще можно понять, и я чувствую себя совершенно неспособным создать хороший компилятор для Ады. Но даже если пренебречь затратами труда по созданию излишне больших систем и стоимостью памяти для хранения их кода, то настоящие затраты скрыты в невиди-

мых усилиях бесчисленных программистов, безуспешно пытающихся понять эти программы и эффективно их использовать.

Другой общей характеристикой упомянутых мною проектов был *выбор инструментария*. По моему мнению, инструментарий должен быть соизмерим с изделием; он должен быть по возможности прост, но не проще того. На деле выбор инструментария может привести к обратным результатам, когда большая часть всего проекта заключается в изготовлении этого инструментария. В проектах Эйлер, Алгол-W и PL360 большое внимание уделялось разработке табличных методов синтаксического анализа «снизу вверх». Позднее я снова вернулся к простому методу рекурсивного спуска «сверху вниз», который безусловно более понятен и при продуманно выбранном синтаксисе достаточно мощен. При разработке аппаратного обеспечения Лилит мы пользовались лишь хорошим осциллографом, и только изредка у нас возникала необходимость в логическом анализаторе. Это было возможно благодаря относительно систематизированной, свободной от искусственных ухищрений концепции процессора.

Каждый отдельно взятый проект прежде всего являлся *обучающим экспериментом*. Лучше всего учиться, изобретая что-либо. Только непосредственно в *процессе разработки* проекта я смог в достаточной мере познакомиться с внутренне присущими ему трудностями и приобрести уверенность в том, что с деталями можно совладать. Я никогда не мог отделить друг от друга процессы проектирования и реализации языка, ибо стремление жестко определить язык, пренебрегая обратной связью с конструированием его компилятора, кажется мне самонадеянным и непрофессиональным. Поэтому я принимал участие в создании компиляторов, в схемотехнике, в разработке текстовых и графических редакторов, и, как следствие, в микропрограммировании, в системном программировании на языках высокого уровня, занимался расчетом схем, компоновкой плат и даже связыванием проводов в жгуты. Это может показаться странным, но я просто люблю делать практическую работу собственными руками значительно больше, чем руководить группой. Кроме того, я понял, что исследователи более охотно признают руководителей, непосредственно входящих в рабочую группу, чем специалистов по организации труда, будь то управляющий в промышленности или профессор университета. Я стараюсь не забывать, что обучение на хороших примерах часто является самым эффективным, а иногда и единственно возможным методом. И, наконец, каждый из этих проектов был осуществлен благодаря энтузиазму и желанию преуспеть: все сотрудники знали, что они занимаются стоящим делом. Это, возможно, самое необходимое, но также и наиболее труднодостижимое

условие успеха. Я был счастлив иметь сотрудников, которые позволили себе увлечься проектом, и я хочу воспользоваться этой возможностью, чтобы поблагодарить их всех за существенный вклад, который они внесли в нашу работу. Я признателен всем, кто так или иначе участвовал в разработках: непосредственно в нашей группе или же помогая нам, тем, кто проверял наши результаты и сообщал о своих впечатлениях, подавал новые идеи, критикуя или одобряя нашу работу, тем, кто создавал общества пользователей. Без них ни Алгол-W, ни Паскаль, ни Модула-2, ни Лилит не стали бы тем, чем они являются сейчас. Эта премия Тьюринга высоко оценивает также и их вклад.

# Введение к части II

## КОМПЬЮТЕРЫ И МЕТОДЫ ИХ ИСПОЛЬЗОВАНИЯ

В части II настоящей книги помещено двенадцать очерков. Одиннадцать из них представляют собой перепечатки Тьюринговских лекций в том виде, в каком они были первоначально опубликованы в изданиях АСМ, а двенадцатый — обширный постскрипtum к одной из никогда не публиковавшейся Тьюринговской лекции.

Следует отметить, что все авторы в той или иной степени включили в свои лекции личные наблюдения и впечатления. Два лектора, Морис Уилкс и Дж. Уилкинсон, были лично знакомы с Аланом Тьюрингом и воспользовались возможностью поделиться интересными и поучительными воспоминаниями о Тьюринге как ученом и товарище по работе.

Лекция Мориса Уилкса «Компьютеры прежде и теперь», прочитанная в 1967 г., является обзором достижений вычислительной математики того времени, и в 1987 г. ее название следовало бы понимать примерно так: «Компьютеры тогда и перед тем». Это не должно вызывать затруднений у тех, кто знаком с программными циклами («в следующий раз это значение времени будет значением времени прошлого раза»). Когда Уилкс проводит аналогию между проблемами разработки компьютеров «тогда» и сходными трудностями, связанными с разделением времени «теперь», то читатель, для которого работа в режиме разделения времени — привычный образ жизни, может сообразить, чему сейчас это соответствует: например, доступу к базам данных по сетям связи или разработке способа соединения микрокомпьютера с большой ЭВМ.

Проницательные соображения Уилкса о совместимости аппаратуры с программным обеспечением и его комментарии о роли языков программирования и структур данных, о важности графических средств и управления процессами, а также о существенных качествах переносимости и параллелизма звучат весьма современно. А глядя на нынешний энтузиазм по поводу экспертных систем, основанных на методах искусственного интеллекта, трудно удержаться от цитирования его замечания: «Животные и машины построены из совершенно разных материалов и на совершенно разных принципах».

Хотя Уилкс уделяет основное внимание вовсе не аппаратуре, стоит отметить, что он вместе с авторами двух других ста-

тей (Хэммингом, а также Ньюэллом и Саймоном) особо указывает на уместность слова «machinery» в названии общества, выступающего спонсором этих лекций — это тем более интересно в свете упорно продолжающихся многолетних попыток изменить название «Association for Computing Machinery» на что-то более «соответствующее».

Четыре другие работы посвящены общим вопросам вычислений и информатики и представляют разнообразные подходы к этим вопросам.

Ричард Хэмминг в лекции 1968 г., «One Man's View of Computer Science», изложил характерный для него прагматический подход. Хотя Хэмминг, вероятно, известен прежде всего как изобретатель носящих его имя кодов, исправляющих ошибки, его взгляд прикладного математика, занявшегося программированием в те годы, когда численные вычисления преобладали, выражен часто цитируемым высказыванием: «Цель вычисления — понимание, а не числа». Его мнения о необходимости математического образования для подготовки специалистов по информатике, о том, где преподавать прикладные разделы информатики (на соответствующих этим приложениям факультетах, а не на факультете информатики), а также о важности обучения «стилю» программирования, полностью согласуются с современными тенденциями, и ознакомиться с его доводами полезно и сейчас. Его описание различия между чистой и прикладной математикой (которое несколько лет назад вызвало критику со стороны важных фигур из числа чистых математиков в колонке писем в журнале Science) также должно привлечь внимание в наше время повышенного интереса к «теоретическим» вопросам информатики. Наконец, его замечания о «деликатных вопросах этики» в том, что связано с компьютерами, ныне не менее актуальны, чем тогда, когда он их писал.

В лекции Марвина Минского 1969 г. «Форма и содержание в информатике» исходным пунктом служит то, что автор называет смещением формы и содержания в трех областях: теории вычислений, языках программирования и образовании. Его замечания, относящиеся к первым двум из этих областей, можно с пользой для себя прочесть вместе с «теоретическими» работами, обсуждаемыми ниже, и работам по языкам части I соответственно. Более половины лекции, однако, посвящено образованию, и наблюдения Минского о математическом образовании детей, отражающие его совместную работу с Сеймуром Пэйпертом, постоянно критикуют «новую математику». Хотя увлечение последней пошло на убыль по сравнению с 1969 г., отмеченные в лекции примеры озабоченности формой в ущерб содержанию по-прежнему актуальны.

В лекции 1975 г. Аллена Ньюэлла и Херберта Саймона

«Информатика как эмпирическое исследование: символы и поиск» природа символьных систем и эвристического поиска обсуждается с философской точки зрения применительно к информатике вообще, но более конкретно в контексте искусственного интеллекта. Ньюэлл и Саймон считают фундаментальными символическое обозначение объектов и символическую интерпретацию процессов, причем и объекты, и процессы принадлежат реальному миру. Поэтому эвристический поиск выдвигается на первый план в качестве главного метода получения решений «задач», включающих объекты и процессы, а способы организации такого поиска зависят от понимания характера объектов и процессов (отсюда «эмпирическое исследование»). Ньюэлл и Саймон применяют эти идеи к программам, способным решать задачи «на уровне экспертов — профессионалов» (в момент написания работы таких программ было мало), «на уровне компетентных любителей» и так далее.

В лекции Кеннета Айверсона 1979 г. «Нотация как средство мышления» обсуждаются общие свойства систем нотации, такие как «содержательность» и «экономичность», и показывается, чем эти качества полезны при решении задач. В качестве примера нотации взят язык программирования APL, что неудивительно, так как автор является изобретателем этого языка. Язык APL находит применения во многих разделах математики, например в алгебре, теории чисел, теории графов. Такая широта также неудивительна для тех, кто знает, что диссертация Айверсона была посвящена численному анализу, а руководил ей экономист, и что он занимался исследованием операций и был соавтором книги по автоматической обработке данных. На первый взгляд этот подход кажется прямо противоположным подходу Ньюэлла и Саймона, будучи чрезвычайно «аналитическим» в отличие от «эвристического», но если рассматривать анализ как средство сужения дерева поиска до предельного случая только одной возможности, а именно до прямого решения, то эти два подхода можно считать двумя сторонами одной медали.

Семь других лекций части II посвящены более специальным вопросам теории и методам применения компьютеров. Хотя принято классифицировать практику программирования, деля ее на два класса, системное и прикладное программирование, несколько лет назад появился новый термин «методология программирования». Он обозначает промежуточную область методов, «выведенных из широких предметных областей применения компьютеров, обладающих одинаковой структурой, процессами и методами». Эта цитата взята из рекомендаций для учебных программ по информатике (Communications of the ACM, март 1968). Среди областей, перечисленных в этом

документе, есть численный анализ, обработка данных и управление файлами, а также искусственный интеллект. Хотя трудно утверждать, что эта терминология завоевала всеобщее признание, она отражена в основных чертах в современной классификации журнала *Computing Reviews*, содержащей специальный раздел «Методологии программирования», а также две другие рубрики, представляющие важные темы этого типа, «Методы вычислений» и «Информационные системы».

Четыре работы из настоящего сборника представляют три упомянутые выше «методологические» области.

Лекция Дж. Уилкинсона 1970 г. «Некоторые замечания специалиста по численному анализу» в основном ретроспективна, но содержит наблюдения о точности численных вычислений — области, в которой автором получены основополагающие результаты. «Неудачи в области матричных вычислений», которые он описывает в деталях (в сущности, это неспособность соответствующих профессиональных групп осознать важность результатов об устойчивости приближенных вычислений), сейчас, по-видимому, преодолены в среде профессиональных вычислителей (исследованиями практики вычислений, разработками математического обеспечения для таких задач и так далее). Аналогичные замечания, однако, могут быть сегодня вполне обоснованно сделаны в отношении практики применения персональных компьютеров и программного обеспечения, поступающего на массовый рынок.

Лекция Джона Маккарти 1971 г. «Общность в системах искусственного интеллекта» (хотя она цитировалась под названием «Современное состояние исследований по искусственному интеллекту») — это, как отмечалось выше, та Тьюринговская лекция, которая ранее не публиковалась. В помещенном здесь постскриптуме автор ретроспективно рассматривает проблему общности — и как она выглядела в 1971 г., и как в последующих работах по искусственному интеллекту ее пытались решить.

Два других очерка относятся к более новой области информатики, а именно к управлению данными. Это лекции Чарлза Бахмана 1973 г. «Программист как навигатор» и Э. Ф. Кодда 1981 г. «Реляционная база данных: практическая основа эффективности». То, что эти два автора представляют противоположные позиции в споре между реляционным и сетевым подходами к базам данных, продолжающимся в этой области несколько лет, не должно затемнять того факта, что обе работы, каждая по-своему, дают полезные идеи об управлении данными как общей проблеме информатики.

Бахман, в частности, описывает в рамках метафоры «навигация» способы доступа к данным, хранимым в структуриро-



ванном и связанном перекрестными ссылками виде. Если стоит критиковать эту точку зрения, то, вероятно, за описание этой деятельности как обязанности исключительно программиста, что вводит в заблуждение, так как в свете современных тенденций с базами данных должны взаимодействовать не только программисты, но и «конечные пользователи». В обширном постскриптуме к этой статье Бахман возлагает на программиста еще более широкий круг обязанностей.

Кодд, написавший свой очерк через восемь лет после Бахмана, представляет обзор таких основных мотивировок своего подхода, как независимость данных, коммуникабельность и возможность обработки множеств, а также описывает реляционную модель и ее функционирование на основе базовых операций выбора, проекции и соединения. За эти годы реляционный подход превратился из предмета академического интереса в практический метод. Автор обсуждает желательность подъязыка, способного функционировать в двух режимах (интерактивном и «встроенном» в прикладную программу). Такой язык манипулирования данными действительно полезен, поскольку многие обсуждения систем с базами данных переоценивают роль интерактивного режима и склонны забывать о том, насколько важно для таких информационных систем запрограммированное взаимодействие с хранимыми базами данных.

Три остальные работы посвящены вычислительной сложности, которая была важным объектом теоретических исследований в течение нескольких последних лет. Лекции Микаэля Рабина 1976 г. «Сложность вычислений», Стефена Кука 1982 г. «Обзор вычислительной сложности» и, наконец, Ричарда Карпа 1985 г. «Комбинаторика, сложность и случайность» в совокупности дают прекрасный обзор всей этой области в доступной неспециалисту форме.

Рабин описывает, какого рода задачи ставит перед собой теория сложности, тогда как Кук, пораженный тем, «как много было сделано в этой области» после лекции Рабина, дает обстоятельный отчет об исследованиях сложности задач в нескольких проблемных областях. В этих двух работах, вместе взятых, содержится более 90 ссылок на литературу. Наконец, Карп предлагает глубокое обсуждение некоторых важных направлений в контексте его собственной работы и работы его коллег в этой области. Все эти три работы вместе с дополнительным интервью с Карпом позволяют читателю почувствовать себя присутствующим при разработке этих фундаментальных проблем.

Роберт Л. Эшенхёрст  
Чикаго, Иллинойс

1967

## Компьютеры прежде и теперь

*М. В. Уилкс*

Кембриджский университет  
Кембридж, Англия

Воспоминания о ранних разработках, приведших к появлению крупных электронных компьютеров, показывают, что для создания и внедрения в практику первых мощных и инженерно проработанных компьютеров потребовалось значительно больше времени, чем ожидалось. Замечания о современном состоянии компьютерной проблематики обосновывают необходимость дальнейших разработок.

Я не думаю, что многие из последующих тьюринговских лауреатов смогут похвастаться личным знакомством с Аланом Тьюрингом. Прославившая его работа о вычислимых числах была опубликована в 1936 г., когда компьютеров еще не существовало. Впоследствии Тьюринг стал одним из первых среди целой плеяды выдающихся математиков, внесших свой вклад в разработку и применение компьютеров. Он был весьма яркой фигурой на заре разработки цифрового компьютера в Англии, и было бы затруднительно не упоминать о нем, рассказывая об этом периоде.

### ВРЕМЕНА ПЕРВООТКРЫВАТЕЛЕЙ

Самое важное событие в моей жизни произошло в 1946 г., когда я получил телеграмму с приглашением прослушать в конце лета того же года учебный курс по компьютерам в филладельфийской школе Мура по электротехнике. Мне удалось прослушать этот курс, хотя и не с самого начала, и он произвел на меня сильнейшее впечатление. Ничего подобного никогда раньше не было, а о достижениях школы Мура и других начинателей компьютерной техники тогда знали лишь немногие. Курс слушали 28 человек из 20 организаций. В роли основных преподавателей выступали Джон Мочли и Проспер Экерт. Они находились на гребне успеха, создав первую электронную вычислительную машину ЭНИАК, работа которой, впрочем, основывалась не на принципе хранения программы в машинной памяти. Размер этой машины впечатляет даже теперь: она содержала примерно 18 000 электронных ламп. Хотя машина ЭНИАК весьма успешно и очень быстро справлялась с вычислением баллистических таблиц, для чего она и предназначалась,

но ее применение в качестве универсального вычислительного устройства наталкивалось на ряд серьезных ограничений. Во-первых, программа вводилась с помощью штекеров, гнезд и переключателей, и поэтому требовалось много времени для перехода от одной задачи к другой. Во-вторых, емкость внутренней памяти была рассчитана на хранение только 20 чисел. Экерт и Мочли сознавали, что основная проблема связана с памятью, и предлагали применять для будущих машин ультразвуковые линии задержки. При этом команды и числа находились бы смешанно в одной и той же памяти, привычным теперь для нас образом. Когда эти новые принципы были провозглашены, стало видно, что более мощные, чем ЭНИАК, компьютеры можно построить с использованием десятой части оборудования, затраченного на ЭНИАК.

В то время фон Нейман сотрудничал со школой Мура в качестве консультанта, но лично познакомиться с ним мне удалось лишь несколько позже. Вычислительная техника очень многим обязана фон Нейману. Он сразу осознал перспективы метода, получившего известность под названием «логическое проектирование», а также возможности, заложенные в принципе запоминаемой программы. Весьма существенно, что для поддержки этих новых идей фон Нейману пришлось воспользоваться своим громадным авторитетом и влиянием, поскольку некоторым они казались слишком революционными и раздавались громкие возгласы о том, что ультразвуковая память окажется недостаточно надежной и что смешивание команд и чисел в одной памяти противостоит естественности.

Последующие события убедительно подтвердили принципы, которым Экерт и Мочли научили в 1946 г. тех из нас, кому посчастливилось прослушать этот курс. Впрочем, в начале пятидесятых годов возникали затруднения. Разумеется, первые работоспособные компьютеры с запоминаемыми программами представляли собой лабораторные модели. Они не были в достаточной степени инженерно проработаны, и в них не в полной мере использовались технологические возможности того времени. Потребовалось неожиданно много времени для создания и внедрения в практику первых более мощных и полностью инженерно проработанных компьютеров. В ретроспективе этот период кажется не слишком длительным, но тогда это было время отчаянных поисков и даже взаимных обвинений.

В прошлом году я часто ощущал, что мы проходим весьма сходный этап в отношении разделения времени. Эта разработка влечет за собой много далеко идущих последствий относительно связей между компьютерами, с одной стороны, и отдельными пользователями и их сообществами — с другой, и она разожгла воображение многих людей. Прошло уже не-

сколько лет после того, как были продемонстрированы первые системы. И снова уходит неожиданно много времени на переход от экспериментальных систем к высокоразвитым, полностью воплотившим в себе современный уровень технологических возможностей. В результате настал период неуверенности и недоуменных вопросов, весьма напоминающий прежний период, о котором я упоминал. Когда все эти проблемы окажутся в прошлом, мы быстро забудем испытания и огорчения, которые теперь переживаем.

В ультразвуковых запоминающих устройствах было принято запоминать до 32 слов впритык друг к другу в одной и той же линии задержки. Частота импульсов была весьма высока, но все же пользователей очень раздражала пустая трата времени на ожидание, пока придет нужное слово. Поэтому большинство компьютеров на линиях задержки проектировались так, что программист, наловчившись, мог размещать свои команды и числа в памяти таким образом, чтобы минимизировать время ожидания. Сам Тьюринг был первооткрывателем этого способа логического проектирования. Позднее сходные методы применялись для компьютеров с памятью на магнитном барабане, и на основе этих методов расцвела общая теория так называемого *оптимального кодирования*. Я чувствовал, что этот род человеческой изобретательности бесперспективен в долгосрочном плане, поскольку рано или поздно мы должны были получить запоминающее устройство с поистине произвольным доступом. Поэтому нам в Кембридже совсем не пришлось заниматься оптимальным кодированием.

Профессия математика не помешала Тьюрингу проявлять определенный интерес к инженерной стороне проектирования компьютеров. В 1947 г. обсуждался вопрос о том, нельзя ли найти для ультразвуковых линий задержки материал, более дешевый, чем ртуть. Тьюринг внес в это обсуждение свежую струю, предложив использовать джин, который, по его словам, содержит алкоголь и воду именно в такой пропорции, чтобы обеспечить нулевой температурный коэффициент скорости прохождения при комнатной температуре.

Причина первоначальных успехов состояла в том, что группы в разных частях света готовились конструировать экспериментальные компьютеры, совсем необязательно стремясь сделать их прототипами для серийного производства. В результате формировались основы знаний о том, что работоспособно, а что неработоспособно, что выгодно делать, а что невыгодно. Хотя рассмотрение коммерчески доступных в настоящее время компьютеров не дает оснований считать, что эти уроки усвоены, несомненно, что такая широта фронта первоначальных исследований окупалась с лихвой. Я считаю важным, что-

бы такая широта фронта исследований имелась и теперь, когда мы учимся конструировать мультипрограммные микропроцессорные вычислительные системы с коллективным доступом. Вместо того чтобы собирать компоненты и электронные лампы для изготовления компьютера, мы должны теперь учиться собирать модули памяти, процессоры и периферийные устройства для изготовления системы. Я надеюсь, что найдутся деньги для конструирования больших систем только с исследовательскими целями.

Многие ранние разработки цифровых компьютеров выполнялись в университетах. Несколько лет назад было широко распространено мнение, что университеты уже сыграли свою роль в проектировании компьютеров и что теперь эту заботу можно благополучно предоставить промышленности. Я не считаю необходимым, чтобы проектированием компьютеров занимались все университеты, но меня радует, что некоторые из них сохранили активность в этой области деятельности. Помимо своих традиционных функций распространения знаний и сохранения открытой для общества информации, которая в противном случае могла бы быть скрыта, университеты в состоянии внести особый вклад, поскольку они не связаны коммерческими соображениями и, в частности, свободны от необходимости следовать моде.

## ХОРОШИЙ И ПЛОХОЙ ЯЗЫКИ

Постепенно утихли споры относительно проектирования самих компьютеров, и мы стали обсуждать достоинства и недостатки изощренных приемов программирования. Началась битва за автоматизацию программирования, или, как мы сказали бы теперь, за использование языков программирования высокого уровня. Я хорошо помню свое участие в спорах на эту тему на одном из первых совещаний АСМ примерно в 1953 г. Выступал и Джон Карр, который разделил программистов на две группы. Первая включала в себя «примитивов», которые полагали, что все команды следует писать в восьмеричном, шестнадцатеричном или некоем сходном виде и не хотели тратить время на то, что они называли чудными схемами. Вторая состояла из «прогрессистов», считавших себя пионерами новой эры автоматизации программирования. Я поспешил отнести себя к прогрессистам, хотя, помнится, предостерегал против надежд на интерпретирующие системы, бывшие тогда в моде, и высказывался в пользу компиляторов. (Сомневаюсь, чтобы термин «компилятор» тогда широко использовался, хотя он был уже введен Грейс Халпер.)

Серьезные возражения против автоматизации программирования должны были касаться эффективности. Компилированные программы не только работали медленнее, чем закодированные вручную, но, что имело тогда большее значение, они требовали больше памяти. Другими словами, для выполнения такой же работы нужен был более мощный компьютер. Все мы знаем, что, несмотря на справедливость этих возражений, они не оказались решающими и что люди пошли на эту жертву ради достижения автоматизации программирования. В сущности, без этого оказалось бы невозможным впечатляющее расширение вычислительной техники за последние несколько лет. Теперь ведется весьма сходная дискуссия относительно разделения времени, и возникающие возражения против такого разделения очень похожи на те возражения, которые ранее выдвигались против автоматизации программирования. И снова я нахожусь на стороне прогрессистов и надеюсь, что нынешние дебаты приведут к аналогичному результату.

Иногда я опасаюсь, что в дискуссии относительно автоматизации программирования симпатии Тьюринга определенно оказались бы на стороне «примитивов». Изобретенная им система программирования для первого компьютера в Манчестерском университете была в высшей степени изощренной. Он сам обладал исключительно гибким мышлением и не видел необходимости делать уступки менее сообразительным людям. Я помню, как он решил, — вероятно, потому, что кто-то показал ему последовательность импульсов на осциллооскопе, — что нужно писать двоичные числа в обратном порядке, с последней значащей цифрой слева. При случае он распространял этот подход и на десятичную нотацию. Я хорошо помню, как однажды во время лекции, когда он перемножал на доске некоторые десятичные числа, чтобы проиллюстрировать тезис о проверке программ, все мы оказались не в состоянии следить за его мыслью, пока не поняли, что он пишет числа задом наперед. Не думаю, что он шутил или пытался посрамить нас; просто он не мог представить себе, что такой пустяк может так или иначе сказаться на чем-то понимании сути дела.

Я полагаю, что через двадцать лет люди будут вспоминать период, в котором мы теперь живем, как время, когда только начиналось понимание основных принципов проектирования языков программирования. Меня огорчает, когда я слышу от вполне разумных людей мнение, что настало время выбрать в качестве всеобщего стандарта один или два языка. Нам действительно нужны временные стандарты для ориентации, но нам не следует рассчитывать на достижение стабильности еще по крайней мере некоторое время.

## СИНТАКСИС БОЛЕЕ ВЫСОКОГО УРОВНЯ

Заметным достижением последних нескольких лет явилось существенное углубление понимания синтаксиса и синтаксического анализа. Оно привело к практическим успехам в конструировании компиляторов. Ранним достижением в этой области, не получившим в свое время должной оценки, был компилятор Брукера и Морриса.

Теперь люди начали понимать, что не все проблемы являются по своей природе лингвистическими и что настало время уделять больше внимания способам хранения данных в компьютерах, т. е. структурам данных. В своей прошлогодней Тьюринговской лекции А. Перлис привлек внимание к этой тематике. В настоящее время выбор языка программирования эквивалентен выбору структуры данных, и если эта структура не подходит для тех данных, которые вы хотите обрабатывать, то остается только глубоко посочувствовать вам. В некотором смысле представляется более логичным сначала выбрать подходящую для проблемы структуру данных, а затем искать или конструировать с помощью набора имеющихся средств язык, пригодный для манипулирования такой структурой данных. Люди иногда толкуют о языках программирования высокого и низкого уровней, не вполне отчетливо формулируя, что они при этом имеют в виду. Если язык высокого уровня таков, что структура данных фиксирована и не поддается изменениям, а в языке низкого уровня имеется некое разнообразие выбора структур данных, то я полагаю, что мы можем наблюдать отход к языкам программирования низкого уровня, по крайней мере для некоторых целей.

Впрочем, здесь я сделал бы замечание. В языке высокого уровня значительная часть синтаксиса и большая часть компилятора относятся к механизму формулирования описаний, к формированию составных операторов из простых и к казуистике условных операторов. Все это совершенно не зависит от того, что делают операторы, которые фактически работают с данными, или на что похожи структуры данных. В сущности, мы имеем два языка, один внутри другого. Внешний язык относится к управлению последовательностью действий, а внутренний язык оперирует данными. Можно было бы иметь стандартный внешний язык — или небольшое количество таких языков на выбор — и ряд внутренних языков, которые могли бы вкладываться в них. В случае необходимости, чтобы учесть особые обстоятельства, можно было бы сконструировать новый внутренний язык. При его встраивании во внешний он пользовался бы преимуществами мощных средств, обеспечиваемых внешним языком, для организации последовательности вычис-

лений. Когда я думаю о различных специальных языках, потребность в которых мы начали ощущать, — например, для управления в реальном времени, для машинной графики, написания операционных систем и др., — мне все больше кажется, что нам следует принять некую систему, которая избавила бы нас от необходимости затрачивать силы на разработку и освоение нового внешнего языка для каждой новой ситуации.

Фундаментальная важность структур данных может быть проиллюстрирована на примере рассмотрения проблемы проектирования единого языка, который оказывался бы предпочтительным как для чисто арифметической работы, так и для символьных манипуляций. Попытки создать такой язык оказались разочаровывающими. Трудность в том, что в этих двух случаях для эффективной реализации требуются совсем разные структуры данных. Возможно, нам следует признать эту трудность непреодолимой и отказаться от поисков такого языка общего пользования, который обеспечивал бы все для всех.

Существует тенденция развития программного обеспечения, которой, по-видимому, уделяется незаслуженно мало внимания. Речь идет о повышении мобильности, т. е. легкости переноса языковых систем с одного компьютера на другой. Долгое время имелась возможность обеспечивать такую мобильность посредством написания системы целиком на некотором широко используемом языке программирования высокого уровня, например на Алголе или Фортране. Однако этот метод заставляет применять структуры данных, свойственные языку, на котором написана система, что налагает очевидные ограничения на эффективность.

Для того чтобы систему можно было сразу переносить с одного компьютера на другой, не прибегая к фиксированному языку-посреднику, она должна быть написана в первую очередь в машинно-независимой форме. Здесь не место углубляться в различные приемы для переноса подходящим образом сконструированной системы. Они, в частности, включают начальную загрузку и использование примитивов и макроопределений. Часто обеспечение переноса включает в себя выполнение некоторой работы на компьютере, на котором система уже работает. Х. Хаски на самом раннем этапе успешно работал в этой области с системой Neliac.

Есть основания надеяться, что вновь найденная мобильность распространится на операционные системы или по крайней мере на их значительные части. В общем, мне кажется, что наступает новый период, когда в меньшей степени будет ощущаться неудобство несовместимости основных машинных кодов. Эта тенденция будет нарастать за счет расширения применения внутренних файловых систем, в которых информация



может храниться в системе в алфавитно-цифровой, т. е. полностью машинно-независимой форме. Тогда сможем без всяких затруднений присоединять к нашим компьютерным системам группы устройств, которые теперь считаются в принципе несовместимыми. В частности, я полагаю, что в больших системах будущего не обязательно все процессоры должны будут поставаться одним и тем же производителем.

## ПРОЕКТИРОВАНИЕ И СБОРКА

Последние несколько лет характеризовались усиленным вниманием к машинной графике. Думаю, что специалисты по информатике уже давно понимали полезность при определенных обстоятельствах графических средств общения с компьютером, но для многих из нас явился неожиданным интерес к этой проблеме со стороны инженеров-механиков. Обычно инженеры общаются между собой с помощью чертежей и эскизов, и как только они увидели чертежи на экране монитора, многие из них сразу решили, что уже решены все проблемы использования компьютеров в инженерном проектировании. Разумеется, мы знаем, что дело обстоит совсем не так и что придется затратить много тяжкого труда, прежде чем удастся реализовать все возможности графического представления информации. Однако первая реакция инженеров показала нам два обстоятельства, о которых не следует забывать. Во-первых, по мнению инженеров-проектировщиков, обычные средства общения с компьютером оказываются совершенно неподходящими. Во-вторых, такая иная форма графического представления информации является чрезвычайно важной для современного машиностроительного конструирования. Мы должны либо обеспечить такую возможность в своих вычислительных системах, либо взять на себя непосильную задачу выучить новую породу инженеров, которым будет присущ другой способ мышления.

Имеются признаки того, что нынешний бум машинной графики будет сопровождаться соответствующим возрастанием интереса к компьютерному управлению объектами. Уже началась разработка нескольких таких проектов. Они в значительной степени стимулируются модой на искусственный интеллект. Эта мода отражается и в выбираемых задачах, и в применяемой стратегии программирования.

Впрочем, мои личные интересы в этой области являются более прагматичными. Я считаю, что компьютерно управляемые механические устройства имеют большое будущее на заводах и повсеместно. Автоматизация изготовления деталей машин уже достигла впечатляющих масштабов, и появление станков с числовым программным управлением сделало возможным

автоматическое изготовление весьма сложных деталей относительно малыми партиями. Напротив, гораздо более скромными являются успехи в автоматизации сборки деталей и узлов в готовые изделия.

Методы искусственного интеллекта могут оказаться менее подходящими для решения задач проектирования автоматизированных сборочных устройств. Животные и машины строятся из совсем разных материалов и на совершенно различных принципах. Когда инженеры пытаются почерпнуть вдохновение из изучения способов деятельности животных, они обычно идут по ложному пути. Это со всей очевидностью иллюстрируется историей ранних попыток конструировать летающие машины с машущими крыльями. По моему мнению, в самом недалеком будущем мы увидим компьютерно управляемые конвейерные линии, вдоль которых будут располагаться ряды автоматов, управляемых той же самой вычислительной системой. Я думаю, что эти автоматы будут скорее напоминать современные станки, а не пальцы руки, хотя они станут не такими громоздкими и будут существенно использовать обратную связь от различных датчиков.

## СЛЕДУЮЩИЙ ПРОРЫВ

Думаю, что все мы задаемся вопросом, останутся ли компьютеры такими, какими мы их теперь знаем, или же они претерпят радикальные изменения. При обсуждении этого вопроса хорошо было бы точно выяснить, чего мы уже достигли. Принятие принципа, что процессор в каждый момент времени занимается только одним делом, по крайней мере с точки зрения программиста, сделало программирование концептуально очень простым и проложило путь для последовательного усложнения, уровень за уровнем, чему мы все были свидетелями. Наблюдения за попытками программировать на первых компьютерах, в которых умножения и другие операции выполнялись параллельно, привели меня к мнению, что важность упомянутого принципа трудно переоценить. Что касается аппаратного обеспечения, тот же принцип привел к разработке систем, в которых высокий коэффициент использования аппаратуры поддерживался применительно к самой разной проблематике, т. е. к разработке поистине универсальных компьютеров. Вычислительная машина ЭНИАК, наоборот, содержала уйму аппаратуры, часть которой была предназначена для вычислений, а часть — для программирования, но применительно к средне-статистической задаче только сравнительно небольшая доля этой аппаратуры использовалась в любой конкретный момент времени.

Если произойдут революционные сдвиги, они должны быть сопряжены с внедрением высокой степени параллелизма, который станет возможным благодаря применению интегральных схем. Проблема состоит в том, чтобы добиться достаточно высокого коэффициента использования аппаратуры, потому что без этого параллелизм не принесет нам большей эффективности. Сильно запараллеленные системы зачастую оказываются эффективными только применительно к тем задачам, которые имел в виду проектировщик. Для других задач наблюдается тенденция к падению коэффициента использования аппаратуры до столь низкого уровня, что при длительном счете обычные компьютеры оказываются более эффективными. Я думаю, что для сильно запараллеленных систем нам неизбежно придется принимать более высокую степень специализации, ориентированной на конкретные области прикладных задач, чем принято в настоящее время. Безусловно, важным фактором является абсолютная стоимость интегральных схем, но следует заметить, что значительное снижение такой стоимости оказалось бы благоприятным и для компьютеров с традиционной архитектурой.

Существует проблематика, в которой, по моему мнению, мы должны возлагать особые надежды на высокую степень параллелизма. Это распознавание образов в двух измерениях. Современные компьютеры оказываются удручающе неэффективными при решении таких задач. Здесь я имею в виду не только распознавание рукописных букв. Многие проблемы символьных манипуляций включают в себя значительный элемент распознавания образов, хорошим примером может служить синтаксический анализ. Я не исключаю возможности того, что произойдет большой концептуальный прорыв в области распознавания образов, который революционным образом преобразует всю проблематику машинных вычислений.

## РЕЗЮМЕ

Я занимался разными разделами информатики с ее ранних дней по нынешнее время. Я начинал не с самого начала, потому что первооткрыватели работали не с электронными, а с механическими и электромеханическими устройствами. Тем не менее мы очень обязаны им, и я полагаю, что даже теперь их работы полезно изучать.

Рассматривая ретроспективно, как менялись интересы специалистов по вычислительной технике и программированию и как постоянно расширялся круг пользователей вычислительных систем, нельзя не удивиться способности компьютеров связывать подлинной взаимной заинтересованностью людей с самы-

ми разнообразными исходными целями и побуждениями. Именно этому мы обязаны жизнестойкостью и энергией нашей Ассоциации. Если даже сочтут необходимым изменить ее название, я надеюсь, что сохранятся слова «вычислительная техника» или некий общепонятный синоним. Ибо всех нас связывает не какая-то абстракция наподобие машины Тьюринга или информации, а реальная аппаратура, с которой мы повседневно работаем.

# Одна из точек зрения на информатику

*Р. В. Хэмминг*

Белл телефон лабораториз, Инк.  
Мюррей Хилл, Нью Джерси

В последнее время многие специалисты склоняются к тому, что информатика должна уделять больше внимания практическим вопросам. Технический аспект имеет важное значение потому, что большинство из возникающих в настоящее время трудностей связаны не с теоретической возможностью сделать что-либо, а скорее с практическим вопросом, каким образом это можно сделать лучше и проще.

Преподавание информатики могло бы стать более эффективным, если внести в него некоторые изменения, например, включение в учебные планы практикума по программированию, уделение большего внимания нематематическим дисциплинам, больше практического программирования и меньше абстрактной теории, больше серьезности и меньше игры.

Разрешите мне сначала сказать несколько слов о своих личных впечатлениях. Когда человеку сообщают, что ему присуждена премия Тьюринга, то сначала это его удивляет, особенно если премия присуждена нетеоретику. Через некоторое время удивление сменяется радостью. Несколько позже возникают вопросы. «А почему я? Почему на фоне всего того, что сделано и делается в области вычислений ЭВМ, выделили меня и мою работу?» Мне кажется, это происходит ежегодно с кем-нибудь, а в этом году счастье выпало мне. В любом случае разрешите поблагодарить вас за оказанную мне честь. Это признание относится также и к компании «Белл телефон лабораториз», где я работаю и которая создала все условия для достижения успеха.

Я выбрал для своей лекции тему «Одна из точек зрения на информатику», поскольку вопрос: «Что такое информатика?» постоянно обсуждается специалистами. К тому же во введении к блестящему докладу «Программа-68» [1] отмечается: «Комитет надеется, что продолжающийся диалог по процессу и целям обучения информатике останется первостепенным в ближайшие годы». Наконец, неправильно полагать, что Тьюринг, в честь которого названы наши ежегодные доклады, занимался исключительно машинами Тьюринга; в действительности он внес значительный вклад во многие направления этой науки и наверняка заинтересовался бы данной темой, хотя, возможно, и не совсем тем, о чем я буду говорить.

Вопрос: «Что такое информатика?» в настоящее время звучит в различных формах, среди которых основными являются:

«Что такое информатика сегодня?», «Во что она может развиться?», «Во что она должна развиваться?», «Во что она разовьется?».

Ни на один из этих вопросов невозможно дать точный ответ. Много лет назад один знаменитый математик написал книгу «Что такое математика?» и нигде даже не попытался дать определение математике, он просто написал книгу о математике. И даже если вы время от времени будете находить более четкое определение некоторых разделов математики, все равно ее наиболее принятым определением останется: «Математика — это то, чем занимаются математики», за которым следует другое: «Математики — это люди, которые занимаются математикой». То, что является правильным в определении математики, является правильным и в определении других наук: чаще всего не существует ясного, четкого определения отдельной научной области.

Сталкиваясь с подобными трудностями, многие, в том числе иногда и я, чувствовали, что не следует принимать участия в таких дискуссиях, нужно просто продолжать *работу* в этой области. Однако, как хорошо отметил Джордж Форсайт в своей недавней статье [2], «имеет значение, как информатику определяют в Вашингтоне, округ Колумбия». По его словам, там склоняются к мнению, что она является частью прикладной математики, и поэтому обращаются к математикам за советом о распределении финансовых средств. Примерно так же обстоит дело повсюду; и в промышленности, и в высших учебных заведениях можно увидеть, где и как зарождалась вычислительная техника: и в электротехнике, и в физике, и в математике, и даже в области бизнеса. Естественно, что представления людей об этой области науки могут в значительной мере повлиять на ее дальнейшее развитие. Таким образом, хотя мы и не можем надеяться на окончательное разрешение этого вопроса, нам необходимо чаще обновлять и пересматривать наши взгляды на то, чем является предмет нашего обсуждения и чем он **должен стать**.

Во многих отношениях мне было бы удобнее поговорить об узкой методической стороне информатики — во всяком случае, это было бы проще. Однако мне хотелось бы особо подчеркнуть опасность потеряться в деталях предмета, особенно в ближайшее время, когда ожидается настоящий потоп научных статей. Мы должны уделять достаточное внимание разносторонности подготовки специалистов, причем в условиях, когда возрастает необходимость специализации для получения тем диссертационных работ, публикации множества статей и др. Мы обязаны готовить студентов к 2000 году — году, когда многие из них достигнут вершины в своей профессиональной деятельности. Мне

кажется, что выражение «специализация — путь к тривиальности» более всего подходит именно к информатике.

Я уверен, вы знаете, что объем наших научных знаний удваивается каждые 15—17 лет. Но я более чем уверен, что в информатике темпы его роста сейчас намного выше, во всяком случае, они были более высокими в течение последних 15 лет. Мы должны учитывать такой прирост информации во всех наших планах и признать, что мы стоим на пороге появления «почти бесконечного» объема знаний. Во многих отношениях классическое представление о том, что ученый знает около 90% всей информации, связанной с предметом его исследований, сегодня уже неверно. Все более узкая специализация не может стать решением этой проблемы, так как часть трудностей состоит в быстро растущем взаимодействии отраслей знаний. По моему личному мнению, нам необходимо обращать больше внимания на качественную сторону материала, чем на количественную, и учитывать, что внимательный, критический, хорошо продуманный научный обзор может гораздо больше повлиять на развитие нашей области, чем новый второстепенный материал.

Мы живем в мире серых полутонов, но для обсуждения проблемы (или, вернее, даже для ее обдумывания) часто необходимо четкое разделение на «черное» и «белое». Правда, поступая так, мы грешим против истины, но иначе мы, по всей видимости, не сможем двигаться вперед. Я полагаю, что большую часть противопоставлений в моем докладе вы будете рассматривать именно в этом свете. Честно говоря, я сам в каком-то смысле не верю в их абсолютную истинность, но мне кажется, нет другого такого же простого способа обсуждения этой темы.

В качестве примера разрешите мне привести условное различие между фундаментальными и прикладными исследованиями: фундаментальная наука изучает, *что* является возможным, а что — нет, в то время как прикладная наука занимается *выбором* из множества возможных путей *такого*, который бы соответствовал многим, часто плохо сформулированным экономическим и практическим целям. Мы называем наш предмет «информатикой», но мне кажется, что точнее было бы назвать его «компьютерной инженерией» (computer engineering), если бы не существовало вероятности неправильного толкования такого названия. Большей частью мы не подвергаем сомнению возможность существования монитора, алгоритма, планировщика или компилятора, скорее мы занимаемся поиском практически работоспособного технического решения с разумными затратами времени и усилий. Хотя я и не стану менять название «computer science» на computer engineering, я все-таки

хотел бы, чтобы в преподавание этой дисциплины вносилось больше практического, прикладного, чем мы обычно находим в учебных программах.

Существует и другая причина выделения практической стороны. Предугадывая, насколько это возможно, будущее нашего предмета, я предвижу, что скоро мы будем нуждаться в больших средствах. Но, как известно, общество обычно, хотя и не всегда, охотнее выделяет средства на те цели, которые дают практическую отдачу, чем на цели, которые оно рассматривает как непрактичную деятельность, занятные игры и т. д. Если мы хотим получить крупные суммы, в которых несомненно будем нуждаться, нам необходимо найти практическое применение нашему предмету. Многие из вас заметили, что мы уже заработали себе плохую репутацию во многих областях. Конечно, есть некоторые исключения, однако все вы знаете, как плохо до сих пор мы удовлетворяли потребности в программном обеспечении.

В основе информатики лежит техническое устройство — вычислительная машина. Без нее почти все, чем мы занимаемся, превратится в пустое словословие, едва ли отличающееся от знаменитой схоластики средневековья. Основатели АСМ ясно высказались, что все то, чем мы занимались или чем мы должны были заниматься, основывалось на этом техническом устройстве, и они намеренно ввели в название своей ассоциации термин «машина» (*machinery*). Есть специалисты, которым бы хотелось исключить этот термин с целью символического освобождения нашей сферы деятельности от реальности, однако до сих пор их усилия не увенчались успехом. Я не сожалею о первоначальном выборе и сейчас еще полагаю, что для нас очень важно признать тот факт, что компьютер — машина, обрабатывающая информацию — лежит в основе нашей сферы деятельности.

Каким образом мы сумеем привнести практический аспект, о котором я говорю, а также восстановить нашу репутацию в глазах общества и дать ему то, что необходимо в данный момент? Может быть, наиболее значительным направлением является то, которое мы избрали в нашей практической деятельности и преподавании, хотя и проводимые нами исследования также будут иметь большое значение. Крайне важно избежать пустословия и игры, которым часто предаются «чистые» математики. Прав или неправ «чистый» математик, утверждая, что то, что сегодня кажется совершенно бесполезным, завтра может пригодиться? Учитывая сегодняшнюю ситуацию, я очень сомневаюсь в его правоте. Это не тот путь, который помог бы получить крупные средства, так необходимые для дальнейшего развития нашей науки. Нельзя считать информатику похо-



жей на математику: основным критерием приемлемости должен быть опыт реального мира, а не эстетические соображения.

Если бы мне пришлось составлять программу преподавания нашего предмета, я бы сделал больший, чем в «Программе-68», упор на лабораторную работу и, в частности, обязал бы любого работающего в этой области специалиста, аспиранта или студента посещать лабораторные занятия, где они должны разработать, написать и отладить программу достаточного размера, а также составить документацию. Эта программа может быть моделирующим устройством или упрощенным компилятором для определенного компьютера. О результатах можно было бы судить по стилю программирования, практической эффективности, количеству технических дефектов и по документации. Если хотя бы один из перечисленных этапов работы выполнен плохо, я бы не считал такого кандидата успешно сдавшим курс. При оценке подобной работы необходимо делать четкое разграничение между сообразительностью и настоящим пониманием предмета. Сообразительность была существенна в прошлом, но в настоящее время ее совершенно недостаточно.

Я потребовал бы также введения дополнительного курса еще в одной серьезной области помимо информатики и математики. Без опыта работы на компьютере над реальной задачей наш выпускник может знать все о замечательном инструменте, за исключением того, как им пользоваться. Такой специалист является всего-навсего техником, обладающим большими навыками умелого манипулирования инструментом, но не знающим, как и когда использовать компьютер по его истинному назначению. Я считаю, что настала пора прекратить подготовку «ученых дураков» — у нас более чем достаточно «компьютерщиков». Сейчас нам нужны профессионалы!

Необходимость программирования реальных задач признана и в «Программе-68» в следующей форме: «Эта цель может быть достигнута путем организации летней практики, совмещения работы и обучения, введения неполного рабочего дня в компьютерных центрах, организации специальных курсов или каким-либо другим подходящим способом». Я считаю, что таким способом могут стать лабораторные курсы с жесткой программой под вашим собственным контролем, а все вышеприведенные предложения Комитета навряд ли будут эффективными или достаточными.

Возможно, наиболее неоднозначным вопросом в составлении учебных программ по подготовке специалистов по информатике является включение в них математических курсов. Многие из нас пришли в эту область, имея солидную математическую подготовку, и поэтому считаем, что такая подготовка необходи-

ма всем специалистам. Слишком часто учитель пытается сделать из ученика свою собственную копию. Однако нетрудно заметить, что в прошлом многие настоящие специалисты в области программного обеспечения не имели соответствующей математической подготовки, хотя большинство из них были прирожденными математиками (в смысле того, чем математика является в действительности, а не того, как ее зачастую преподают).

В прошлом я считал, что требование глубоких математических знаний при подготовке исключит из информатики большую часть лучших специалистов. Но с возрастанием значимости планирования и распределения ресурсов компьютеров я вынужден был пересмотреть свою точку зрения. Хотя и очевидно, что частично это будет решаться аппаратными средствами, трудно предположить, что в ближайшее время, как минимум, в течение пяти лет, программисту не потребуется много заниматься планированием и распределением ресурсов. Если это действительно станет характерным, то мы будем вынуждены серьезно рассмотреть вопрос о подготовке специалистов. Если мы не будем давать такой подготовки, то наш специалист по информатике придет к пониманию того, что он является просто «техником», который всего лишь программирует то, что ему заказывают другие. Более того, программистское мастерство, которое раньше мы считали большим достижением, часто зависело от сообразительности и ловкости программиста и почти не требовало математических знаний. Кажется, этот период уже проходит, и если мы хотим, чтобы наши выпускники были способны внести серьезный вклад в науку, мы обязаны дать им хорошую математическую подготовку.

История показывает, что лишь относительно небольшая часть людей в возрасте после 30 лет, не говоря уже о более старшем возрасте, может действительно освоить новое в математике; таким образом, если в будущем математика будет играть значительную роль, мы должны дать учащимся соответствующую математическую подготовку еще в школе. Мы, конечно, можем временно обойти это решение, предложив учащимся два параллельных курса: один с изучением математики, другой без ее изучения, предупредив, однако, что второй путь ведет в тупик, по крайней мере в отношении последующего университетского обучения (предполагая, что мы убеждены в первостепенном значении математики для обучения на факультете информатики).

Признав необходимость основательной математической подготовки, мы тут же сталкиваемся с еще более сложной задачей — понять, какие именно курсы нам нужны. Несмотря на многочисленные утверждения теоретиков о фундаментальной

важности математики, необходимо признать, что мы используем сравнительно малую ее часть на практике. Однако, по моему мнению, необходимо, чтобы каждый специалист в области информатики прослушал хотя бы один математический курс. Но, видимо, все дело в том, что организация курсов формальной математики не соответствует сегодняшним требованиям нашего предмета. Нам нужны абстрактная алгебра, теория массового обслуживания, статистика, включая планирование экспериментов, основы теории вероятности, возможно с некоторыми элементами цепей Маркова, отдельные вопросы теории информации и кодирования, кое-что о скорости передачи сигналов, некоторые разделы теории графов и т. п. Однако, как хорошо известно, наша область быстро изменяется, и, возможно, завтра нам могут понадобиться теория функции комплексного переменного, топология и другие дисциплины.

Как я уже говорил, планирование математических курсов является наиболее сложной частью программы. После долгих размышлений на эту тему я пришел к выводу, что если мы хотим, чтобы наши выпускники внесли значительный вклад в науку и не опускались до уровня техников, умеющих, по мнению других специалистов, только управлять инструментом, лучше дать им слишком много математики, чем слишком мало. Я очень хорошо понимаю, что такой подход может исключить из нашей области многих из тех, кто в прошлом содействовал ее развитию, и этот вывод меня совсем не радует, однако он именно таков. Дальнейшее развитие информатики требует владения математикой.

Наиболее часто учебные программы по информатике упрекают в том, что они почти полностью игнорируют практическую сторону и язык Кобол. Я полагаю, что вопрос о том, нужно или нет преподавать на факультете информатики практическое применение компьютеров или язык Кобол, связан отнюдь не с их прикладным значением. Скорее, как мне кажется, вопрос заключается в том, может ли факультет административного управления выполнить эту работу намного лучше нас и является ли то, что присуще приложению к бизнесу, принципиальным для других аспектов информатики. И то, что говорилось о применении компьютеров к бизнесу, надо полагать, относится и к другим областям их применения, которые можно преподавать на других факультетах. Я глубоко уверен, что с теми ограниченными ресурсами, которыми мы располагаем в данный момент и которыми будем располагать еще в течение длительного времени, мы не должны пытаться преподавать практические применения компьютеров на нашем факультете, скорее их нужно вводить на тех факультетах, где и предполагается их использование.

Проблема роли аналоговых вычислений в учебной программе факультета информатики отличается от подобной проблемы, стоящей перед применением компьютеров в специальных областях, так как нет другого места, где она могла бы решаться. Нет ни малейшего сомнения в том, что аналоговые вычислительные машины экономически важны и будут оставаться такими еще некоторое время. Но нет также ни малейшего сомнения и в том, что класс таких машин, включая даже гибридные компьютеры, не обладает в настоящее время теми интеллектуальными возможностями, которые присущи цифровому вычислению. Кроме того, сущность хорошего аналогового вычисления заключается в понимании физических ограничений оборудования и в специфическом искусстве масштабирования, особенно по временной переменной, что достаточно чуждо остальной области информатики. Таким образом, налицо тенденция скорее к игнорированию аналогового вычисления, чем к отказу от него. Его или не преподают, или делают факультативным, и это, вероятно, лучшее, что можно сделать в настоящее время, когда в центре внимания находится универсальный цифровой компьютер.

В настоящее время чувствуется «игровой» налет в программах многих наших курсов информатики. Мне постоянно приходится выслушивать от друзей, которые хотели бы взять на работу хороших знающих специалистов в области программного обеспечения, что те, кто к ним обращается, не представляют для них интереса как специалисты. По их мнению, наши выпускники в основном стремятся играть в игры, писать хитроумные программы, которые на самом деле не работают, трюковые программы и так далее, но неспособны сконцентрировать свои усилия таким образом, чтобы то, что они обещают, было сделано вовремя и в практически пригодной форме. Если бы я услышал эту жалобу лишь однажды от друга, воображающего себя заправским инженером-практиком, я бы ею пренебрег; к сожалению, я слышал это неоднократно от многих компетентных, разумных и понимающих людей. Как я уже говорил, поскольку мы так отчаянно нуждаемся в финансовой поддержке для текущей работы и будущего расширения наших возможностей, нам следовало бы подумать, как избежать таких замечаний о наших выпускниках в ближайшие годы. Собираемся ли мы продолжать выпускать «продукцию», отвергаемую столь многими, или же мы собираемся выпускать ответственных, работоспособных специалистов, в которых действительно нуждается наше общество? Я полагаю, что необходимость выбора последней альтернативы становится все более ясной, и отсюда мое внимание к практическим аспектам информатики.

Одна из причин, по которым наши выпускники больше заинтересованы в «красивом» программировании, чем в практических результатах, состоит в том, что многие наши учебные курсы преподаются людьми с привычками и навыками «чистых» математиков. Чистый математик начинает с поставленной задачи или с некоего выбранного им варианта постановки этой задачи и получает то, что он называет ее решением. В прикладной математике необходимо добавить два критически важных шага: во-первых, исследование связи между математической моделью и реальной ситуацией и, во-вторых, связи между результатами, выдаваемыми этой моделью, и реальной ситуацией (или, если вам угодно, интерпретация этих результатов в терминах реальной ситуации). Вот в этом-то и заключается резкое различие. Человек, занимающийся прикладной математикой, должен быть готов принять на себя ответственность, заявив: «Если вы сделаете то-то и то-то, вы с высокой точностью увидите то-то и то-то, и поэтому оправданно продвигаться вперед и тратить время и средства указанным способом», тогда как чистый математик только пожмет плечами и заметит: «Я за это не отвечаю». Но кто-то же должен взять на себя ответственность за выбор того или иного направления, и мне кажется, что тот, кто берет на себя такую ответственность, должен получить больше доверия, чем это обычно у нас бывает. Таким образом, в процессе преподавания мы обязаны особенно подчеркивать необходимость принятия на себя полной ответственности за решение *всей* проблемы, а не только чисто математической ее части. Это и есть еще одна причина, почему я обратил внимание на техническую сторону различных предметов и постарался приуменьшить чисто математические аспекты.

Трудность, конечно, заключается в том, что большинство преподавателей в информатике являются чистыми математиками, а также в том, что преподавать чистую математику гораздо легче, чем прикладную. Относительно немного преподавателей способны вести данный предмет именно так, как мне представляется необходимым. Это означает, что мы должны делать все возможное с помощью тех средств, которыми располагаем. С осторожностью выбирать необходимое нам направление, прививать, где это возможно, свойственные практику чувство ответственности и прагматизма, а не только умение доказывать существование решения.

Очень жаль, что на ранних этапах развития информатики для достижения успеха решающее значение имели талант и способность справляться с громадным объемом мелочей. Но если мы хотим, чтобы студент вырос в специалиста, способного справляться с более обширными разделами информатики, то

он должен иметь и развивать другие способности, которые не используются и не тренируются на начальных этапах обучения. Многие наши выпускники никогда не делают этого второго шага. Примерно так же обстоят дела и в математике: в первые годы обучения требуется тривиальное владение арифметическими операциями и формальными символическими операциями школьной алгебры, но в высшей математике для успеха необходимы совсем другие способности. Как я уже говорил, многие программисты, сделавшие успешную карьеру в области, в которой самую важную роль играют мелочи, не смогли развить более широких способностей, и они продолжают преподавать и распространять свою науку о мелочах. На более высоких уровнях информатики требуется не «черно-белое» мышление, характерное для большей части математики, но способность к взвешенным субъективным суждениям, учитывающим противоречия между поставленными целями, что характерно для прикладных технических дисциплин.

Пока что я обрисовал программирование как дисциплину, для которой, выражаясь словами одного из моих друзей, свойственно изобретение специальных трюков для каждого конкретного случая при отсутствии общих принципов. В такой оценке программирования как трюкачества так много правды, что затруднительно обсуждать вопрос о том, чему следует учить на курсах программирования. Столь многое из того, что мы сделали, сделано специально для конкретных случаев, и мы находились под таким давлением требований получить хоть что-то работающее как можно быстрее, то у нас теперь есть лишь немного ценного, способного выдержать придирчивый взгляд ученого или инженера, спрашивающего: «В чем состоит содержание науки о программном обеспечении?» Сколь многочисленны трудные для понимания идеи в этой области! Сколь многое представляет собой просто нагромождение одной мелочи за другой без всякого тщательного анализа! И если компиляторы, содержащие 50 тысяч слов, могут быть позже заменены новыми, всего лишь в пять тысяч слов, то как далеки от разумного были первоначальные решения!

Я далеко не эксперт в области программного обеспечения, и мне сложно сделать серьезные предположения по поводу того, что необходимо предпринять в этой области, и тем не менее я чувствую, что очень часто мы довольствовались таким низким уровнем качества, что сами себе нанесли серьезный урон. Кажется, что мы неспособны использовать машину, которая, по нашему общему мнению, является мощным инструментом управления и преобразования информации, для решения наших собственных задач в области программного обеспечения. У нас есть компиляторы, ассемблеры, мониторы и т. п., но

только для других, и в то же время, анализируя повсеместную работу программиста, я часто удивляюсь тому, насколько мало он использует машину в своей собственной работе. У меня накопилось достаточно опыта в дискуссиях со специалистами в области программного обеспечения, чтобы настаивать на обучении использованию машин практически на всех стадиях нашей работы. Очень немногие из системных программистов вообще пытаются использовать компьютер в своей работе. Существует множество ситуаций, когда использование компьютера может оказать программисту огромную помощь. Я припоминаю один случай, когда неспециалист, имевший большую программу на языке Фортран, составленную на стороне, хотел приспособить ее для нашего (местного) использования и составил простую программу на Фортране для локализации всех операторов ввода-вывода и всех библиотечных ссылок. Мой личный опыт показывает, что большинство программистов предпочитают просматривать данные распечатки текстов программ, отыскивая нужные им строки, и, конечно, сначала пропускают некоторые из них. Мне кажется, необходимо убедить программистов, что компьютер является их самым мощным подручным средством и что они должны как можно чаще использовать его в своей работе, особенно при необходимости просмотра длинных списков символов, вместо того чтобы выполнять эту операцию вручную, как это практикуется у нас повсеместно. Если все то, о чем я говорю сейчас, действительно верно, значит, мы упустили этот момент в нашей прежней работе со студентами. Конечно, наиболее подготовленные из них используют компьютер именно так, как я рекомендую, однако, по моим наблюдениям, большинство выпускников-программистов совершенно не пользуются им.

Образно говоря, наши сегодняшние методы преподавания программирования сводятся к тому, что мы даем первокурсникам учебник грамматики и словарь и говорим им, что отныне они великие писатели. Мы практически не знакомим их со стилем программирования. И действительно, до сих пор я не встречал ни учебника стилистики, ни «Антологии программ». Программирование — такое же сложное искусство, как и литературное творчество. И там и здесь предпочтение отдается краткости. Анализируя метод обучения хорошему литературному языку — написание сочинений, составление докладов и выступлений, по которым оцениваются успехи студентов, — мы сразу заметим, что в преподавании программирования подобный подход отсутствует. К сожалению, лишь немногие программисты действительно обладают тем, что я называю «стилем», и стремятся создавать настоящие произведения искусства в программировании. В результате лишь немногие из них

владеют изящным, «поэтическим» стилем, большинство же пишут в грубой, тяжелой «прозе».

Я очень сомневаюсь в том, что стиль программирования тесно связан с каким-либо определенным машинным языком, как и в том, что литературный стиль какого-то определенного естественного языка может действительно значительно отличаться от литературного стиля в другом языке. Конечно, в разных языках существуют какие-то определенные способы выражения мыслей, однако основные нормы хорошего стиля одинаковы для большинства западноевропейских языков, во всяком случае, для тех из них, которые я знаю. И я думаю, что все это верно и для сегодняшних цифровых машин.

Конечно, когда я утверждаю, что в образование необходимо внести больше от инженерного, чем от научного творчества, меня могут неправильно понять. Поэтому я должен пояснить, что пришел в информатику, защитив докторскую диссертацию по чистой математике. Несмотря на то что я постоянно призываю обращать больше внимания на практические занятия и развитие инженерного дара, я тем не менее отдаю себе отчет в том, что в силу отсутствия необходимых нам теорий мы не до конца понимаем то, чем в действительности занимаемся.

Действительно, одним из наших слабых мест, по моему мнению, является то, что если Ньютон мог сказать: «Я видел немного дальше, чем другие, потому что стоял на плечах гигантов», то сегодня мы вынуждены признать, что стоим на ногах друг у друга. Быть может, основной проблемой, стоящей перед нами, является необходимость перехода к такой ситуации, когда мы могли бы продолжить работу предыдущего поколения, а не делать ее заново. Ведь все-таки наука должна накапливать знания, а не бесконечно повторять одно и то же.

Это рассуждение подводит меня к еще одному различию, — различию между ненаправляемым поиском и фундаментальными исследованиями. Все хотели бы заниматься ненаправляемым поиском, и большинству хотелось бы верить, что именно такой тип исследований и является фундаментальным. Мне кажется, что фундаментальными следует считать такие исследования, на результатах которых будет строиться работа других исследователей. В конце концов, какой другой смысл можно вложить в понятие «фундаментальные»? Я полагаю, и опыт это подтверждает, что лишь весьма немногие ученые способны проводить фундаментальные исследования. И хотя невозможно с полной уверенностью сказать заранее, станет ли данная работа фундаментальной, можно довольно точно предсказать ее дальнейшую судьбу. Занимаясь этим вопросом, я пришел к заключению, что возможности той или иной работы превратиться в фундаментальное исследование зависят не столько от



проблемы, которую она решает, сколько от подхода к ее решению.

Численные методы являются единственной частью нашей учебной программы, которую принимают практически все, так как они имеют хоть какое-то содержание. Тем не менее нас часто и, надо сказать, справедливо упрекают в том, что большая часть нашего учебного материала написана для математиков, и действительно, он содержит больше теоретического, чем практического материала. Причина, конечно, заключается в том, что многие специалисты, работающие в этой области, пришли в нее из математики, отчасти так и остались математиками и до сих пор подсознательно подходят ко всему с точки зрения математики. Я уверен, что многие из вас знакомы с моими возражениями по этому поводу, и мне, видимо, не нужно повторять их [3].

Многие коллеги говорили мне, и я сам отмечал, что большинство курсов, предложенных в программе преподавания нашего предмета, чаще всего перегружены деталями, за которыми не видно основных идей. Не стоит давать студентам все методы нахождения вещественных нулей функции, лучше показать наиболее типичные и эффективные из них, которые иллюстрировали бы основные концепции численных методов. И то, что я сейчас говорил о численных методах, еще в большей степени относится к курсам программного обеспечения. Мне, как и другим специалистам, кажется, что эти курсы не содержат достаточного количества фундаментальных идей в области программного обеспечения, которые могли бы оправдать то учебное время, которое для них отводится. Из этих курсов необходимо исключить массу ненужных мелочей и оставить только тот материал, который действительно содержит важные идеи.

Разрешите мне сейчас затронуть деликатную тему, — тему профессиональной этики. Много раз отмечалось, что поведение программистов по сравнению с бухгалтерским персоналом во многом оставляет желать лучшего [4]. Кажется, мы не прививаем нашим студентам чувства «неприкосновенности» информации, касающейся людей или частных компаний. Мои наблюдения говорят о том, что программисты лишь поверхностно знакомы с нормами этического поведения. Например, многие из них совершенно уверены в том, что имеют полное право забрать с собой при смене места работы любую программу. Мы должны иметь это в виду и изучить, каким образом основы этического поведения преподаются на курсах бухгалтерского учета, выпускники которых обладают большими навыками в области этики. Мы много говорим об опасности создания крупных банков личных данных на людей, но практически не занимаемся этическим воспитанием наших выпускников.

Далее я хотел бы коснуться такой темы, как нормы профессионального поведения. О них говорилось в недавних публикациях [5], и они показались мне действительно необходимыми, однако я считаю себя вправе вновь задать вопрос, каким образом они включены в программу обучения наших студентов и действительно ли мы способны привить студентам правила такого поведения. Конечно, недостаточно только зачитывать их в аудитории каждый день. Такой путь обучения этическим и профессиональным нормам поведения неэффективен. Совершенно очевидно, что преподаватели других учебных заведений находят способы доводить до студентов нормы профессионального поведения, которые (хотя впоследствии и не все их придерживаются) преподаются лучше, чем у нас. Таким образом, нам необходимо разобраться в их методах и приспособить их к нашим нуждам.

И наконец, разрешите мне перейти еще к одной актуальной теме, которую мы часто затрагиваем, — теме социальной ответственности. Мы часто говорим об этом на встречах, обсуждаем ее в кулуарах, за чашкой кофе и даже в барах, однако я снова задаю вопрос: «Каким образом тема социальной ответственности раскрывается в наших учебных программах?» Тот факт, что мы не имеем разработанных правил для этого, не может служить оправданием ее отсутствия в наших программах.

Я полагаю, что все три темы — этического и профессионального поведения и социальной ответственности — должны быть в обязательном порядке включены в программу подготовки программистов. Мне лично кажется, что введение отдельного курса по этим темам не будет эффективным. Исходя из моего собственного представления о методах привития этих норм студентам, лучше всего они передаются через поведение самих преподавателей. Поучительны неожиданные вещи: форма, в которой даются замечания преподавателя, его манеры и умение держать себя. Таким образом, сам преподаватель должен понять, что значительная часть его деятельности как преподавателя заключается в выборе способов и методов доведения этих деликатных, ускользающих моментов до своих студентов. Преподаватель не имеет права сказать: «Это меня не касается». Об этом должны постоянно говорить все преподаватели или не говорить вообще. Если эти навыки не прививать студентам каким-либо образом, то наша область сохранит свою сегодняшнюю репутацию, которая может удивить вас, если коллеги с других факультетов честно выскажут свое мнение.

В заключение разрешите мне обрисовать истинную перспективу информатики. Это совершенно новая область, находящаяся в постоянном развитии. У нас было очень мало времени для того, чтобы подготовиться и осуществить на практике многое

из того, что мы уже знали раньше. Но по крайней мере в университетах мы многого добились: нам удалось учредить самостоятельные факультеты, на которых действуют курсы с неплохими программами, необходимым оборудованием и соответствующим профессорско-преподавательским составом. Теперь мы хорошо подготовлены и считаем, что именно сейчас необходимо углубить, усилить и улучшить нашу деятельность таким образом, чтобы мы могли гордиться тем, *что мы преподаем, как мы это преподаем*, и студентами, которых мы выпускаем. Мы готовим не просто техников, ученых дураков или «компьютерщиков»; мы отдаем себе отчет в том, что в наше сложное время мы должны готовить специалистов, способных принять на себя ответственность за решение проблем в нашем быстро меняющемся обществе. В противном случае мы должны будем признать, что не состоялись как преподаватели и лидеры в такой интересной и важной области, как информатика.

1969

# Форма и содержание в информатике

*М. Минский*

Массачусетский технологический институт  
Кембридж, Массачусетс

Чрезмерное увлечение формализмом препятствует развитию информатики. Путаница между формой и содержанием обсуждается применительно к трем областям: теории вычислений, языкам программирования и обучению.

Проблема современной информатики состоит в навязчивой озабоченности формой в ущерб содержанию.

Нет, начать нужно не с этого. По любым прежним стандартам жизненная сила информатики огромна. Какая другая сфера интеллектуальной деятельности когда-либо достигала такого расцвета за двадцать лет? К тому же теория вычислений, по-видимому, некоторым образом включает в себя учение о форме, так что эта озабоченность не так уж неправомерна. Тем не менее я утверждаю, что чрезмерное увлечение формализмом тормозит развитие нашей науки.

Прежде чем должным образом углубиться в обсуждение темы, я хочу выразить признательность моим коллегам и студентам, которые способствовали тому, что я удостоился Тьюринговской премии. Комплекс вопросов, некогда философских, а ныне научных, сопряженных с понятием интеллекта, вызывал чрезвычайный интерес у Алана Тьюринга, и он наряду с немногими другими мыслителями — особенно Уорреном Маккаллоком и его молодым сотрудником Уолтером Питтсом — был автором многих ранних исследований, приведших к созданию как самих компьютеров, так и новой технологии, а именно искусственного интеллекта. При оценке этой проблематики факт присуждения данной премии должен привлечь внимание к другим работам членов моего научного коллектива, в первую очередь Р. Соломоноффа, О. Селфриджа, Дж. Маккарти, А. Ньюэлла, Х. Саймона и С. Пейперта, моих ближайших сотрудников по десятилетним исследованиям. В этом эссе нашли отражение взгляды Пейперта.

Данное эссе состоит из трех частей, посвященных недопониманию взаимоотношений между формой и содержанием в *теории вычислений, языках программирования* и в обучении.

## 1. ТЕОРИЯ ВЫЧИСЛЕНИЙ

Чтобы построить теорию, нужно кое-что знать об основных явлениях предметной области. Применительно к теории вычислений нам просто недостает таких знаний, чтобы излагать этот предмет достаточно абстрактно. Вместо этого мы вынуждены подробнее разбирать конкретные примеры, которые понимаем более основательно, и надеяться, что это поможет нам формулировать и обосновывать более общие принципы. Я говорю это не для того, чтобы с ходу принимать как данность многие положения, вероятно истинные, но еще не доказанные. Я полагаю, что многие наши предположения, основанные на так называемом здравом смысле, в действительности ложны. У нас бывают ложные представления о возможных альтернативных вариантах затрат времени и памяти; о компромиссах между временем работы и сложностью программ, между аппаратным и программным обеспечением, между цифровыми и аналоговыми схемами, последовательными и параллельными вычислениями, ассоциативной и адресуемой памятью и о многом другом.

Целесообразно рассмотреть аналогию с физикой, где можно организовать много базовых знаний как набор довольно компактных законов сохранения. Конечно, это всего лишь один из способов описания; можно было бы использовать дифференциальные уравнения, принципы минимума, законы равновесия и т. д. Так, сохранение энергии можно интерпретировать как определение переходов между различными формами потенциальной и кинетической энергии, например между высотой и квадратом скорости или между температурой и давлением — объемом. Можно основывать разработку квантовой теории на балансе между степенью определенности координат и временем или же времени и энергии. В этом нет ничего необычного; всякое уравнение с достаточно гладкими решениями может рассматриваться как некое определение обменов (преобразований) между его переменными. Однако существует много способов формулирования одних и тех же соотношений, и рискованно отдавать чрезмерное предпочтение одной конкретной форме или закономерности и приходить к убеждению, что это *истинный* фундаментальный принцип. (См. на эту тему диссертацию Фейнмана [1].)

Тем не менее распознавание подобных преобразований (об-

менов, компромиссов) часто является зачатием науки, тогда как их количественное выражение — ее рождением. Что в этом смысле мы наблюдаем в области вычислений? В теории рекурсивных функций мы располагаем наблюдением Шеннона [2], заключающимся в том, что любая машина Тьюринга с  $Q$  состояниями и  $R$  символами эквивалентна машине Тьюринга с двумя состояниями и  $nQR$  символами, а также машине Тьюринга с 2 символами и  $n'QR$  состояниями, где  $n$  и  $n'$  — небольшие числа. Таким образом, произведение числа состояний на число символов,  $QR$ , играет роль почти инварианта при классификации машин. К сожалению, это произведение не удастся использовать в качестве полезной меры сложности машины, потому что оно, в свою очередь, является плодом компромисса со сложностью процесса кодирования для машин Тьюринга, а этот компромисс представляется слишком непонятным для того, чтобы его можно было с пользой применить.

Рассмотрим более элементарный, но все же озадачивающий компромисс между сложением и умножением. Сколько умножений требуется для вычисления детерминанта  $3 \times 3$ ? Если мы выпишем полную форму как шесть произведений по три, то понадобится двенадцать перемножений. Если мы соберем множители, воспользовавшись распределительным законом, то число умножений сократится до девяти. Каково минимальное количество, и как доказать это в данном случае и в случае  $n \times n$ ? Важно не то, что мы нуждаемся в ответе. Суть в том, что мы не знаем, как узнать или доказать, что предлагаемые ответы являются правильными! Для одной формулы, возможно, удалось бы организовать некоторый вид исчерпывающего поиска, но на основании этого нельзя было бы установить общее правило. Одной из главных целей исследования должна быть разработка методов доказательства того, что данные конкретные процедуры в тех или иных смыслах являются вычислительно минимальными.

В диссертации Кука [3], в которой используется результат Тоома, сделано удивительное открытие, касающееся умножения; оно обсуждается в публикации Кнута [4]. Рассмотрим обычный алгоритм умножения десятичных чисел: для двух чисел, каждое из которых имеет  $n$  знаков, требуется  $n^2$  произведений однозначных чисел. Обычно полагают, что это минимальное количество. Но предположим, что мы разбиваем числа на две части, так что произведение принимает вид  $N = (@A + B)(@C + D)$ , где @ означает умножение на  $10^{n/2}$ . (Мы пренебрегаем затратами на операцию сдвига влево.) Тогда легко убедиться, что

$$N = @ @AC + BD + @ (A + B) (C + D) - @ (AC + BD).$$

Здесь задействованы только *три* умножения половинной длины вместо четырех, которые, казалось бы, нужны. При большом значении  $n$  такое сведение может очевидным образом применяться снова и снова для меньших чисел. За это приходится платить увеличением количества сложений. Объединив эту и другие идеи, Кук показал, что при любом  $\epsilon$  и достаточно большом  $n$  для умножения требуется меньше чем  $n^{1+\epsilon}$  произведений вместо ожидавшихся  $n^2$ . Аналогичным образом В. Штрассен недавно показал, что для умножения двух матриц  $m \times m$  число умножений можно свести до  $O(m^{\log_2 7})$ , хотя всегда было принято считать, что это число должно быть кубическим полиномом от  $m$ , потому что результат содержит  $m^2$  членов и для каждого из них вроде бы требуется отдельное внутреннее произведение с  $m$  перемножениями. В обоих случаях обычная интуиция длительное время вводила математиков в заблуждение, настолько пагубное, что, по-видимому, никто и не искал лучших методов. Мы все еще не располагаем набором проверенных методов, которые точно устанавливали бы, какова минимальная цена замены умножений сложениями в случае матриц.

Замена умножений сложениями сама по себе может не казаться жизненно важной, но если мы не в состоянии досконально понять нечто столь простое, то нам тем более следует ожидать серьезных затруднений с чем-то более сложным.

Рассмотрим другой компромисс, между размером памяти и временем вычислений. В книге [5] Пейперт и я поставили простую проблему: если имеется произвольный набор слов по  $n$  бит, то сколько обращений к памяти потребуется, чтобы ответить, какое из этих слов является ближайшим (по числу совпадающих бит) к произвольному заданному слову? (Для идентификации *точно* отождествления можно использовать хеширование, и эта проблема достаточно хорошо изучена.) Поскольку существует много способов кодирования «библиотечного» набора слов, причем в одних из них используется больше памяти, чем в других, то более точно проблема формулируется следующим образом: насколько должен возрасти объем памяти для достижения данного сокращения числа обращений к памяти? В определенном смысле ответ тривиален: если память достаточно велика, то требуется только одно обращение, потому что мы можем использовать в качестве адреса сам вопрос и запомнить ответ в регистре с таким адресом. Но если память как раз достаточна для хранения информации в библиотеке, то нужно искать все слова из библиотеки, — и нам не известно сколько-нибудь *значительного промежуточного результата*. Разумеется, это фундаментальная теоретическая проблема восстановления информации, однако, по-видимому, ни у кого нет

каких-либо интересных идей относительно того, как установить хорошую нижнюю границу для этого основного компромисса<sup>1)</sup>.

Другая проблема связана с последовательно-параллельным обменом. Предположим, у нас имеется  $n$  компьютеров вместо всего лишь одного. Насколько мы можем ускорить те или иные виды вычислений? Для некоторых мы можем уверенно получить выигрыш в  $n$  раз. Однако такие вычисления встречаются редко. Для других этот коэффициент составит  $\log n$ , но такие вычисления трудно обнаружить или доказать, какими свойствами они должны обладать. А для большинства вычислений, по моему мнению, нам трудно выиграть хоть что-нибудь; это тот случай, когда имеется множество весьма разветвленных условий, так что попытки предугадывать возможные будущие ответвления обычно оказываются тщетными. Об этом мы не знаем почти ничего. Большинство людей полагают с совершенно неоправданным оптимизмом, что параллелизм, как правило, благоприятствует ускорению большинства вычислений.

Итак, существует несколько плохо понимаемых проблем относительно вычислительных компромиссов. Здесь нет места для обсуждения остальных, например, аналогово-цифровой проблемы. (Некоторые проблемы соотношения локальных и глобальных вычислений намечены в [5].) И мы очень мало знаем о сопоставимости численных и символьных вычислений.

В современных учебных планах по информатике уделяется слишком мало внимания тому, что известно о таких проблемах. Почти все время отводится формальной классификации синтаксических типов языков, пораженческим теориям неразрешимости, фольклору о системном программировании и преимущественно тривиальным фрагментам «оптимизации логического проектирования» — последнее часто в ситуациях, когда практическое искусство эвристического программирования далеко превзошло «теории» для частных случаев, столь громогласно провозглашаемые и неустанно проверяемые, — а также заклинаниям относительно стиля программирования, которые почти наверняка выйдут из моды к тому времени, когда студент завершит свое образование. Даже, казалось бы, наиболее абстрактные курсы по теории рекурсивных функций и формальной логике, по-видимому, игнорируют немногие известные полезные

<sup>1)</sup> В 1964 г. в Ж. вычисл. матем. и матем. физ., т. 4, с. 536—543, В. В. Мартынюк опубликовал статью «Экономная организация поиска и занесения информации при избыточной памяти», в которой предложил алгоритм организации библиотек из  $k$  слов, при котором поиск любого слова осуществляется путем  $i$  обращений к памяти. Этот алгоритм применим при всех значениях  $i=1, 2, 3, \dots, k$ . При любом значении  $i$  для хранения библиотеки требуется примерно  $ik\sqrt[m]{n/k}$  слов (ячеек), где  $m$  — число всех адресуемых ячеек, например  $m=2^n$ . — Прим. перев.



результаты по доказательству свойств компиляторов или эквивалентности программ. В большинстве курсов результаты работы по искусственному интеллекту, некоторым из которых теперь уже исполнилось пятнадцать лет, толкуются как второстепенный набор специальных приложений, хотя на самом деле они представляют одну из важнейших сторон эмпирического и теоретического исследования реальных вычислительных проблем. До тех пор пока все эти предпочтения формы не уступят место вниманию к существенным аспектам вычислений, юному студенту вполне можно посоветовать не обременять себя курсами информатики, учиться программировать, усваивать как можно больше математику и другие фундаментальные научные дисциплины, а также изучать современную литературу по искусственному интеллекту, теориям сложности и оптимизации.

## 2. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Даже в области языков программирования и компиляторов слишком много внимания уделяется форме. Я говорю «даже», потому что можно понять, что это именно та область, в которой форма *заслуживает* преимущественного внимания. Но рассмотрим два утверждения: (1) языки оказываются такими, что в них слишком много синтаксиса, и (2) языки описываются с излишними синтаксическими подробностями.

В компиляторах не уделяется должного внимания значению выражений, вызываний и описаний. Применение контекстно-свободных грамматик для описания фрагментов языков ведет к важным достижениям в единообразии как спецификации, так и реализации. Однако, хотя это вполне оправдывается в простых случаях, попытки распространить этот подход на более сложные области могут затормозить научный прогресс. Возникают серьезные проблемы при использовании грамматик для описания самоизменяющихся или саморасширяющихся языков с применением процессов исполнения наряду со спецификациями. Невозможно описать синтаксически — иначе говоря, статически — допустимые выражения языка, который изменяется. Конечно, должны быть описаны механизмы расширения синтаксиса, однако если они задаются в терминах такого современного языка сопоставления с образцом, как Снобол, Convert [6] или Matchless [7], то не должно быть различия между программой грамматического разбора и описанием собственно языка. Будущие языки программирования станут в большей степени сосредотачиваться на целях и в меньшей степени на процедурах, специфицированных программистом. Дальнейшие суждения являются несколько максималистскими.

Можно обойтись гораздо меньшим объемом синтаксиса, чем обычно считается. Большая часть синтаксиса программирования относится к подавлению скобок или к выражению меток контекста. Существуют варианты, которые используются слишком редко.

Рассмотрим общеизвестный алгоритм извлечения квадратного корня в том виде, как его можно записать на современном алгебраическом языке, пренебрегая такими аспектами, как описания типов данных. Требуется значение квадратного корня из  $A$  при заданном начальном приближении  $X$  и предельно допустимой погрешности  $E$ .

если  $ABS(A - X * X) < E$ , то  $X$  иначе  $SQRT(A, (X + A \div X) \div 2, E)$ .

```
(DEFINE (SQRT A X E)
  (IF(LESS(ABS(-A(*X X)))E)THEN X
    ELSE(SQRT(÷(+X(÷A X))2)E)))
```

Еще предстоит увидеть, является ли синтаксис с явными разделителями косностью или же представляет собой росток будущего. Он обладает значительными преимуществами для редактирования, интерпретации и для *создания программ другими программами*. Полный синтаксис Лиспа можно выучить примерно за час; интерпретатор компактен и не слишком

усложнен, и студенты часто могут отвечать на вопросы о системе, читая саму интерпретирующую программу. Разумеется, это не даст ответов на все вопросы относительно реальной практической реализации, но их не даст и никакой обозримый набор синтаксических правил. Кроме того, несмотря на громоздкость этого языка, многие работающие на переднем крае науки исследователи считают, что он обладает выдающейся выразительной силой. Почти все исследования процедур решения проблем путем построения и модификации гипотез были написаны с применением этого или сходных языков. К сожалению, разработчики языков, как правило, незнакомы с этой областью и склонны отмахиваться от нее как от специального раздела «методов манипуляции символами».

Можно сделать многое для прояснения структуры выражений в таком «синтаксически слабом» языке, воспользовавшись записью текста с различными отступами и другими графическими средствами, которые выходят за рамки собственно языка. Например, можно употребить символ «отсрочки», относящийся к входному препроцессору, чтобы переписать предыдущее выражение как

```

DEFINE (SQRT A X E) ↓ .
  IF ↓ THEN X ELSE ↓ .
    LESS (ABS ↓) E.
      — A (*XX).
    SQRT A ↓ E .
      ÷ ↓ 2 .
      + X(÷ AX)

```

где точка означает «)(», а стрелка означает «вставить здесь следующее выражение, ограниченное точкой и становящееся доступным после замены (рекурсивно) его собственных стрелок». Отступы необязательны. Значительная часть эффекта обычных меток области действия обеспечивается здесь двумя простыми средствами, тривиально реализуемыми программами чтения, и такой текст легко поддается редактированию, потому что подвыражение обычно завершается в каждой строке.

Чтобы ощутить силу и ограниченность операции отсрочки, читателю следует обратиться к своему излюбленному языку и к своим излюбленным алгоритмам и посмотреть, что произойдет. Он обнаружит там много применений для отсрочки и поупражняется в рассуждениях о том, что сказать вначале, какие аргументы выделить и так далее. Разумеется, знак ↓ не решает всех проблем; техника отсрочки требуется также для фрагментов списков, и для этого нужен свой собственный раз-

делитель. В любом случае это всего лишь шаги в направлении более графически выразительных систем описания программ, так как мы не обязаны навсегда ограничиваться только строками символов.

Д. Скотт предложил другое объяснительное средство, состоящее в использовании других скобок для указания функциональной композиции справа налево, так что можно писать  $\langle\langle\langle x \rangle h \rangle g \rangle f$  вместо  $f(g(h(x)))$ , если есть желание показать более естественно, что происходит с величиной в процессе вычисления. Благодаря этому возникает возможность для различных «акцентов», например,  $f(\langle h(x) \rangle g)$  можно прочесть так: «вычислите  $f$  от того, что вы получили после того, как сначала вычислили  $h(x)$ , а затем применили  $g$  к результату».

Пожалуй, это проще объяснить по аналогии, чем на примере. Со своей фанатической приверженностью синтаксису проектировщики языков стали слишком односторонне ориентированы на фразу. Пользуясь такими средствами, как знак  $\Downarrow$ , можно конструировать объекты, более напоминающие абзацы, не возвращаясь полностью к старинным блок-схемам.

Современные языки высокого уровня предлагают слишком мало выразительных возможностей в отношении гибкости стиля. Пользователь не может существенно варьировать последовательность представления мыслей, не меняя самого алгоритма.

## 2.2. ЭФФЕКТИВНОСТЬ И ПОНИМАНИЕ ПРОГРАММ

Для чего предназначается компилятор? Обычно отвечают примерно так: «для трансляции с одного языка на другой» или «чтобы, имея описание алгоритма, превратить его в программу с заполнением многих мелких подробностей». На будущее требуется более честолобивый подход. Большинство компиляторов станут системами, которые «порождают алгоритм по заданному описанию того, что он должен делать». Таковы уж современные системы формирования изображений; они выполняют всю творческую работу, тогда как пользователь только предоставляет образцы желаемых форматов: в этом компиляторы более сведущи, чем пользователи. Хорошими примерами являются также языки сопоставления с образцами. Однако, за исключением немногих частных случаев, проектировщики компиляторов мало преуспели в обеспечении написания хороших программ. Распознавание общих подвыражений, оптимизация внутренних циклов, распределение нескольких регистров и прочее — все это приводит лишь к небольшим линейным улучшениям эффективности, причем даже в этом отношении компиляторы делают очень немного. Можно было бы в зна-

чительно большей степени автоматизировать распределение памяти. Но настоящий выигрыш заложен в анализе *вычислительного содержания* самого алгоритма, а не способа записи этого алгоритма программистом. Например, рассмотрим:

ОПИСАНИЕ  $FIB(N)$ : если  $N=1$  то 1, если  $N=2$  то 1, .  
иначе  $FIB(N-1)+FIB(N-2)$

Это рекурсивное описание чисел Фибоначчи 1, 1, 2, 3, 5, 8, 13... может быть задано на любом приличном языке программирования и приведет к разветвляющемуся дереву шагов вычисления, показанному на рис. 1.

Легко видеть, что количество машинной работы будет возрастать экспоненциально с увеличением  $N$ . (Более точно, оно проходит через примерно  $FIB(N)$  вычислений согласно описанию.) Существуют лучшие способы вычисления этой функции. Так, мы можем описать два временных регистра и вычислить  $FIB(N)$  по формуле

ОПИСАНИЕ  $FIB(NAB)$ : если  $N=1$  то  $A$  иначе  $FIB(N-1A+BA)$

которая является однократно рекурсивной и избавляет от разветвляющегося дерева, или даже можем использовать

$A=0$

$B=1$

ЦИКЛ ЗАГРУЗИТЬ  $AB$

если  $N=1$  вернуть  $A$

$N=N-1$

$B=A+B$

перейти в ЦИКЛ

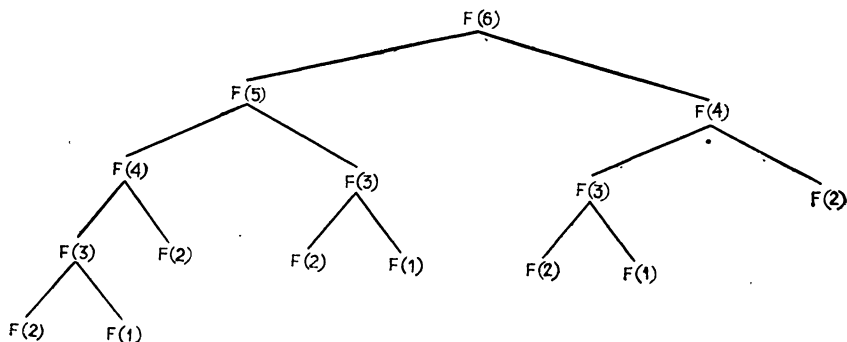


Рис. 1.

Любому программисту быстро придут в голову эти возможности, как только он увидит, что происходит при разветвленном вычислении. Это тот случай, когда «управляемая значениями» рекурсия может быть преобразована в простое повторение. Современные компиляторы не распознают даже простейшие случаи таких преобразований, хотя избавление от экспоненциального роста перевешивает любой возможный выигрыш от локальной оптимизации кода. Нет смысла возражать, что такие выигрыши редки или что подобные преобразования относятся к компетенции программиста. Если важно сэкономить время компиляции, то не следует пренебрегать такими возможностями. Например, для программ, написанных на языках сопоставления с образцом, подобные упрощения действительно часто производятся. Обычно компиляция эффективного дерева грамматического разбора для системы БНФ дает явный выигрыш по сравнению с выполнением примитивного переборного анализа посредством синтеза.

Вне всяких сомнений, систематическая теория таких преобразований трудна. Система должна быть весьма сообразительной для того, чтобы обнаруживать, какие преобразования уместны и когда их применение окупается. Поскольку программист всегда знает свою цель, проблема часто упрощается, если предлагаемый алгоритм сопровождается (или даже заменяется) подходящей декларацией цели работы алгоритма.

Для того чтобы продвигаться в этом направлении, нам нужно знать основы анализа и синтеза программ. В области теории сейчас многое делается для изучения эквивалентности алгоритмов и схем и для доказательства того, что процедуры обладают объявленными свойствами. С практической стороны работы В. Мартина [10] и Дж. Мозеса [11] показывают, как строить системы, оснащенные достаточными знаниями о символических преобразованиях конкретных математических методов для существенной поддержки прикладных математических возможностей пользователей.

Практически ничего не следует из того факта, что проблема сведения программ в общем случае является рекурсивно неразрешимой. Во всяком случае, можно ожидать появления программ, в конечном счете далеко превосходящих человеческие способности к этой деятельности, и можно воспользоваться большим разнообразием преобразований программ в формализованном и отлаженном виде. Непосредственное применение таких преобразований окажется затруднительным. Вместо этого можно ожидать разработок в духе методов, употреблявшихся в символическом интегрировании, например в работах Слэгла [12] и Мозеса [11]. Сначала был разработан набор простых формальных преобразований, соответствующих

элементарным формулам табличных интегралов. На основании этого Слэгл построил ряд эвристических приемов алгебраического и аналитического преобразования практической задачи в эти уже понятые элементы; при этом привлекались характеристики процедуры сопоставления, которые можно назвать «распознаванием образов». В системе Мозеса процедуры сопоставления и преобразования были столь хорошо отработаны, что в большинстве практических задач стратегия эвристического поиска, игравшая важную роль в эффективности программы Слэгла, становилась малосущественным приращением к точному знанию и его искусному приложению, воплощенному в системе Мозеса. Эвристическая система компиляции в конечном счете потребует гораздо больше *общего знания* и здравого смысла, чем системы символьного интегрирования, поскольку цели такой системы ближе к моделированию всей деятельности математика, а не специалиста по интегрированию.

### 2.3. ОПИСАНИЕ СИСТЕМ ПРОГРАММИРОВАНИЯ

Вне зависимости от того, как описывается язык, компьютер должен располагать процедурой его интерпретации. Следует помнить, что *при описании языка главная цель состоит в том, чтобы объяснить, как писать на нем программы и что такие программы означают*. Основная цель *вовсе не состоит* в описании синтаксиса.

В рамках статичной структуры синтаксических правил, нормальных форм, правил подстановок Поста и других подобных схем мы получаем эквиваленты логических систем с аксиомами, правилами вывода и теоремами. Проектирование недвусмысленного синтаксиса соответствует тогда проектированию математической системы, в которой каждая теорема имеет ровно одно доказательство! Но в вычислительном контексте это совершенно не важно. Здесь имеется дополнительный аспект — управление, — который выходит за рамки обычной структуры логической системы. Речь идет о дополнительном наборе правил, которые указывают, когда следует применять то или иное правило вывода. Итак, для многих целей недвусмысленность является надуманной проблемой. Если мы рассматриваем программу как процесс, то можем вспомнить, что наиболее мощными средствами описания процессов являются сами программы, а они по самой своей природе недвусмысленны.

Описание языка программирования через программу не является парадоксом. Разумеется, процедурное описание должно быть понято. Можно достичь этого понимания за счет описания на другом языке, который может быть иным, более знакомым или более простым, чем тот, который описывается. Но

часто бывает, что практичнее, удобнее и правильнее использовать тот же самый язык! Чтобы понять такое описание, нужно знать только работу данной конкретной программы, а не все следствия из всех возможных применений языка. Именно эта конкретизация создает возможность для раскрутки, т. е. способа разработки программного обеспечения, при котором сначала разрабатывается ограниченное подмножество языка, используемое для реализации более обширных его вариантов. Это обстоятельство часто озадачивает начинающих, как, впрочем, и кажущихся авторитетными специалистов.

Применение БНФ для описания формирования выражений может задержать развитие новых языков, которые естественным образом включают цитирование, самомодификацию и манипуляции символами в традиционную алгоритмическую схему. А это в свою очередь тормозит прогресс в направлении систем программирования, ориентированных на целевое решение проблем. Как ни парадоксально, хотя идеи современного программирования разрабатывались из-за трудности изображения процессов в классической математической нотации, тем не менее проектировщики обращаются назад, к ранней форме — уравнению — именно в тех ситуациях, где *нужна* программа. В разд. 3, посвященном обучению, наблюдается сходная ситуация в преподавании, возможно, с еще более серьезными последствиями.

### 3. ОБУЧЕНИЕ, ПРЕПОДАВАНИЕ И «НОВАЯ МАТЕМАТИКА»

Образование — это еще одна область, в которой специалист по информатике путает форму и содержание, но в настоящее время эта путаница неотделима от его профессиональной роли. Он воспринимает свою главную функцию как обеспечение программ и машин для использования в старых и новых методиках обучения. Само по себе это неплохо, но я полагаю, что на нем лежит ответственность за более сложную задачу — вырабатывать и распространять модели самого процесса обучения.

Ниже я кратко обрисую отправную для этого мнения точку зрения (разработанную С. Пейпертом). Следующие утверждения типичны для нашего подхода:

— Помочь людям учиться — это значит помочь им строить в своем сознании различные виды вычислительных моделей.

— Лучше всего с этим справится тот учитель, который имеет в своем сознании разумную модель того, что собой представляет сознание ученика.

— По той же причине студент при отладке своих собственных моделей и процедур должен иметь модель того, что он



делает, и должен знать хорошие приемы отладки, например как формулировать простые, но решающие тестовые примеры.

— Это поможет студенту что-нибудь узнать о вычислительных моделях и программировании. Например, сама идея отладки<sup>1)</sup> является весьма мощной — в отличие от беспомощности, навязываемой традиционными представлениями о дарованиях, талантах и склонностях. Эти представления навязывают человеку вывод: «Я не гожусь для этого», вместо того чтобы поощрить его спросить себя: «Как мне усовершенствоваться в этом?»

Эти утверждения звучат вполне разумно, однако они не относятся к числу основных принципов любой из популярных схем обучения, таких, как «оперативное закрепление», «методы раскрытия», «аудиовизуальный синергизм» и др. Дело не в том, что педагоги игнорировали возможность моделей мышления, а в том, что у них просто не было эффективных способов описывать, строить и проверять такие идеи, пока не началась работа по имитации (компьютерному моделированию) процессов мышления.

Мы не можем снисходить здесь до дискуссий со скептиками, которые считают чрезмерным упрощенчеством (если не нечестивостью или непристойностью) сравнение человеческих умов с программами. Можно отослать многочисленных таких критиков к работе Тьюринга [13]. Тем, кто полагает, что ответ не может храниться в цифровых или каких-либо других машинах, можно возразить [14], что машины, обретя интеллект, вполне возможно, станут полагать точно так же. Некоторые обзоры этой области содержатся в работах Фейгенбаума и Фелдмана [15] и Минского [16]. В этой быстро развивающейся отрасли можно не отстать от жизни, только читая современные диссертации и труды конференций по искусственному интеллекту.

Имеется существенный прагматический довод в пользу наших предложений. Ребенку нужны модели; чтобы понять город, он может использовать модель организма: тот тоже должен есть, дышать, выделять, защищаться и т. д. Не самая удачная модель, но достаточно полезная. Обмен веществ в реальном организме он может понять, в свою очередь сравнив его с машиной. Но для моделирования своей собственной сущности он *не может* воспользоваться ни машиной, ни организмом, ни городом, ни телефонным коммутатором; ему для этого не пригодится ничего, кроме компьютера с его программами и их ошибками. В конце концов в начальном обучении само про-

---

<sup>1)</sup> Тьюринг весьма преуспел в отладке аппаратуры. Он проверял ее, не отключая тока, чтобы не потерять «ощущение» предмета. Теперь так делают все, но это не то же самое в наши дни, когда все схемы работают на трех или пяти вольтах.

граммирование станет играть даже большую роль, чем математика. Тем не менее я выбрал именно *математику* в качестве предмета обсуждения в оставшейся части этой работы отчасти потому, что мы понимаем ее лучше, но главным образом из-за того, что существующее предубеждение против программирования как академической науки привело бы к слишком сильному сопротивлению со стороны аудитории. Я полагаю, что сошел бы и какой-нибудь другой предмет, но выводы и понятия математики являются самыми точными и в наименьшей степени подвержены запутывающему влиянию эмоций.

### 3.1. МАТЕМАТИЧЕСКИЙ ПОРТРЕТ РЕБЕНКА

Представьте себе ребенка в возрасте пяти-шести лет, собирающегося поступать в первый класс. Если мы проэкстраполируем тенденции наших дней, его математическое образование будет проводиться слабо подготовленными учителями и отчасти плохо запрограммированными компьютерами; ни те, ни другие не будут способны сделать намного больше, чем отвечать «правильно» и «ты ошибся»; я уже не говорю, что они будут совершенно не в состоянии разумно интерпретировать то, что ребенок делает или говорит, потому что не будут обладать хорошими моделями ребенка или хорошими теориями детского умственного развития. Ребенок начинает с простой арифметики, теории множеств и начатков геометрии; через десять лет он будет знать немного о формальной теории вещественных чисел, немного о линейных уравнениях, чуть больше о геометрии и почти ничего о непрерывных функциях или предельных переходах. Он станет подростком без особых склонностей к аналитическому мышлению, неспособным применить свой десятилетний опыт к пониманию нового для него мира.

Посмотрим более пристально на нашего ребенка, воспользовавшись составной картиной, извлеченной из работ Пиаже и других наблюдателей умственного развития детей.

Наш ребенок сумеет произнести «один, два, три...» по крайней мере до тридцати, а возможно, и до тысячи. Он будет знать названия некоторых больших чисел, но не сможет увидеть, например, почему десять тысяч равняется сотне сотен. У него будут возникать серьезные затруднения с обратным счетом, если только он недавно не заинтересовался этим. (Хорошее владение обратным счетом облегчит простое вычитание, и имеет смысл несколько попрактиковаться.) У него не особенно развито чувство четности и нечетности.

Он может совершенно надежно сосчитать четыре — шесть предметов, но у него не каждый раз будет сходиться счет пятнадцати рассыпанных предметов. Он будет раздражен этим,

потому что твердо уверен, что должен получать каждый раз одинаковый результат. Поэтому наблюдатель заключит, что ребенок хорошо представляет себе понятие числа, но не столь искусен в его применении.

Впрочем, существенные аспекты его понимания чисел все не будут признаны надежными с точки зрения стандартов, принятых взрослыми. Например, когда предметы переупорядочены перед его глазами, его впечатление об их количестве будет формироваться под влиянием их геометрического размещения. Например, ребенок скажет, что имеется меньше букв *x*, чем букв *y*, в следующих строчках:

x x x x x x x  
y y y y y y y

а когда мы раздвинем буквы *x* и получим

x x x x x x x  
y y y y y y y

он скажет, что букв *x* больше, чем букв *y*. Несомненно, что он при этом отвечает на другой вопрос — о размере; однако это реальный факт: постоянство (сохраняемость) количества в таких ситуациях в малой степени доходит до него. Он не в состоянии эффективно воспользоваться этим фактом в своих рассуждениях, хотя, если его спросить, он продемонстрирует знание того, что количество вещей не может измениться просто из-за их переупорядочивания. Аналогично, когда вода переливается из одного стакана в другой (рис. 2, а), ребенок скажет, что воды в высоком и узком стакане больше, чем в низком и широком. Он плохо оценивает двумерные площади, и поэтому нам не удастся найти такой контекст, в котором ребенок рассматривал бы большую площадь на рис. 2, б как четырехкратное увеличение меньшей площади. Кстати, когда он уже взрослый, то, получив два сосуда, один из которых в два раза превосходит другой по всем измерениям (рис. 2, в), он будет думать, что один вмещает примерно в четыре раза больше жидкости, чем другой; возможно, ему никогда не удастся достичь умения лучше оценивать объем.

Мы мало знаем о том, что происходит в сознании ребенка, когда он думает о числах. Согласно Гелтону [17], тридцать детей из ста будут ассоциировать малые числа с определенными позициями в пространстве перед своим умственным взором, упорядочив их неким специфическим способом, как показано на рис. 3. Возможно, они сохраняют эти ассоциации и в юности и могут пользоваться ими в неясных подсознательных

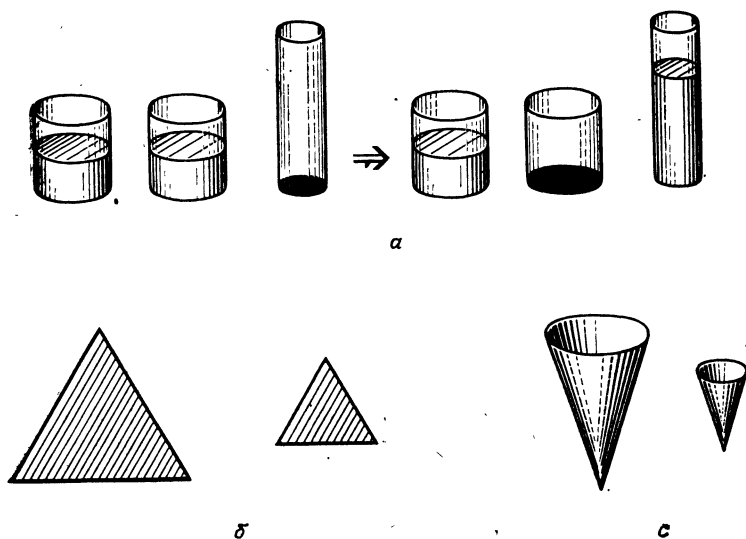


Рис. 2.

процессах запоминания телефонных номеров; а может быть, они сформируют в своем сознании различные пространственно-визуальные представления для исторических дат и т. д. Учитель никогда не почувствует этого, и если ребенок об этом говорит, то даже учитель со своим собственным представлением о числах вряд ли ответит ученику пониманием. (Мой опыт свидетельствует, что требуется ряд тщательно поставленных вопросов, прежде чем один из этих юношей откликнется: «О да! 3 находится над этим местом, чуть сзади».) Когда наш ребенок изучает суммирование столбиком, он может отслеживать переносы цифр в другой разряд, прижимая свой язык к определенным зубам, или использовать некое другое тайное средство временного запоминания, и никто об этом даже не узнает. Возможно, одни способы лучше других.

Детский геометрический мир отличается от нашего. Ребенок не замечает, что треугольники жесткие и этим отличаются от других многоугольников. Он не знает, что аппроксимация окружности стоугольником неотличима от этой окружности, если только она не очень уж велика. Он не начертит куб в перспективе. Он лишь недавно понял, что квадраты становятся

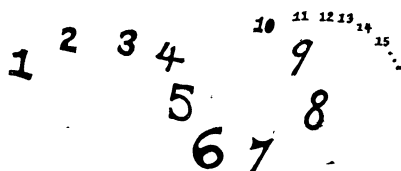


Рис. 3.

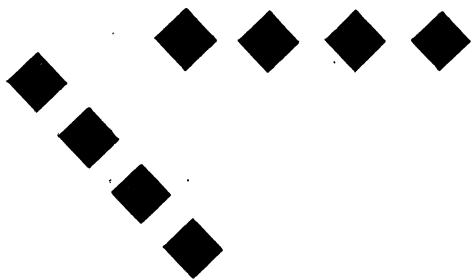


Рис. 4.

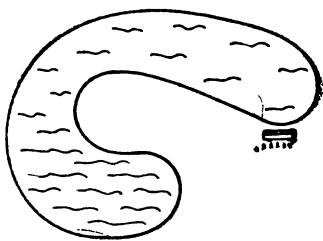


Рис. 5.

ся ромбами, если их повернуть углами вниз. Это перцептивное отличие сохраняется и у взрослых. Так, на рис. 4 мы видим, как заметил Эттниви [18], что восприятие фигур как квадратов или как ромбов зависит от их взаимного расположения на картинке, очевидно, оно определяется нашим выбором осей симметрии, используемых в субъективном описании.

Наш ребенок вполне хорошо понимает топологическую идею замкнутости. Почему же? Это очень сложное понятие классической математики, но в терминах вычислительных процессов оно, по-видимому, оказывается не столь трудным. Но наше дитя почти наверняка будет сбито с толку ситуацией на рис. 5 (см. Пейперт [19]): «Когда автобус начинает экскурсию вокруг озера, мальчик сидит на стороне, удаленной от воды. Окажется ли он на стороне, обращенной к озеру, в некоторый момент экскурсии?» Это затруднение, вероятно, сохранится и на восьмом году жизни ребенка и, по-видимому, связано с его трудностями осознания других абстрактных двойных отрицаний, таких, как вычитание отрицательных чисел или понимание других следствий непрерывности — «в какой точке рейса происходит какое-нибудь внезапное изменение?», — или осознания других мостов между логикой и всеобщностью.

Более детально наш портрет вырисовывается в литературе по психологии развития. Но еще никому не удавалось построить вычислительную модель ребенка, достаточную для того, чтобы увидеть, как эти возможности и ограничения связываются в структуру, совместимую с его столь ярко выраженными способностями к другим делам (а возможно, и логически следующую из этих способностей). Однако такая работа находится в самом начале, и я предполагаю увидеть в следующем десятилетии существенное развитие таких моделей.

Если бы мы больше знали об этих предметах, то смогли бы помочь ребенку. В настоящее время у нас даже нет хорошей диагностики; его явная способность учиться давать правильные ответы на формальные вопросы может свидетельствовать

только о том, что он разработал некоторые изолированные библиотечные подпрограммы. Если они не могут быть вызваны его центральными программами решения проблем из-за использования несовместимых структур данных или по какой-либо причине, мы можем получить проворного решателя тестов, который никогда не будет вполне хорошо мыслить.

До перехода к вычислительным моделям совокупность идей о природе мышления была слишком слабой для того, чтобы поддерживать эффективную теорию обучения и развития. Ни модели с конечным числом состояний бихейвиористов, ни гидравлические и экономические аналогии фрейдистов, ни внезапные озарения гештальт-психологии не дают достаточных оснований для понимания столь затруднительной проблематики. Для этого требуется фундамент из уже отлаженных теорий и решений сопоставимых, но более простых проблем. Теперь же у нас бьют ключом хорошо сформулированные и реализованные идеи для мышления о мышлении, лишь малая часть которых представлена в традиционной психологии:

таблица символов  
простая процедура  
разделение времени  
последовательность вызовов  
функциональный аргумент  
защита памяти  
таблица диспетчирования  
сообщение об ошибке  
прослеживание вызовов функций  
точка прерывания  
языки  
компилятор  
косвенный адрес  
макроопределение  
список свойств  
тип данных  
хеширование  
микропрограмма  
согласование по формату  
замкнутые подпрограммы

магазинный список  
прерывание  
ячейка связи  
общая память  
дерево решений  
компромисс аппаратного и программного обеспечения  
последовательно-параллельный компромисс  
компромисс затрат времени и памяти  
условная точка прерывания  
асинхронный процессор  
интерпретатор  
сборка мусора  
списковая структура  
блочная структура  
упреждение  
оглядка  
диагностическая программа  
управляющая программа

Это лишь немногие идеи, относящиеся к общему системному программированию и отладке; мы ничего не сказали о многих более специфических понятиях из языков, искусственного интеллекта, вычислительной техники и других развитых отраслей информатики. Все они служат сегодня средствами искусно-

го и хитроумного программирования. Но точно так же, как на смену астрологии пришла астрономия, введенная законами Кеплера, открытие принципов эмпирического объяснения интеллектуального процесса в машинах должно привести к возникновению новой науки. (В обучении мы все еще сталкиваемся с таким же состоянием. В разделе комиксов газеты *Boston Globe* содержится астрологическая страница. Помогите справиться с загрязнением интеллектуальной среды!)

Однако вернемся к нашему ребенку. Как наши вычислительные идеи могут помочь ему в связи с его пониманием чисел? В младенчестве он учится распознавать некоторые специальные парные конфигурации, такие, как две руки или два ботинка. Значительно позднее он узнает о некоторых тройках — возможно, с большим затруднением, потому что окружающая среда содержит немного фиксированных троек: если он найдет три пенса, то, наверное, скоро потеряет или проиграет один из них. В конце концов он найдет некую процедуру манипулирования пятью или шестью предметами и выйдет из под власти находок и потерь. Но когда дело касается более чем шести или семи предметов, он останется во власти забывчивости; даже если его словесный счет безупречен, его процедура нумерации будет несовершенна. Он пропустит некоторые элементы, а другие пересчитает дважды. Мы можем помочь, предложив лучшие процедуры; складывание предметов в коробку почти защищает от глупости, поскольку как бы вычеркивает их. Но для неподвижных объектов он по-прежнему будет нуждаться в некой процедуре мысленного группирования.

Прежде всего нужно постараться понять, что делает ребенок, — в этом может помочь слежение за движениями его глаз, — а может быть, достаточно спросить его. Возможно, он выбирает очередной элемент с помощью некоего ненадежного, почти случайного метода, не располагая хорошими средствами для отслеживания того, что уже сосчитано. Можно было бы предложить: *скольжение курсора, изобретение легко запоминаемых групп, нанесение крупных сеток*.

В каждом случае конструкция может быть либо реальной, либо воображаемой. При использовании сеточного метода нужно помнить, что нельзя засчитывать дважды объекты, которые пересекают линии сетки. Учитель должен объяснить, что это хорошо бы планировать заранее, как показано на рис. 6, искривляя сетку, чтобы избежать двусмысленностей.

Математически важный принцип состоит в том, что «любая правильная процедура счета порождает то же самое число». Ребенок поймет, что правилен любой алгоритм, который (1) *подсчитывает все предметы*, (2) *не засчитывает ни один из них дважды*.

Возможно, это процедурное условие кажется слишком простым; даже взрослый может понять его. Во всяком случае, это не та концепция числа, которая принята в том, что теперь носит общее название «новая математика» и преподается в наших начальных школах. Ниже эти вопросы обсуждаются полемически.

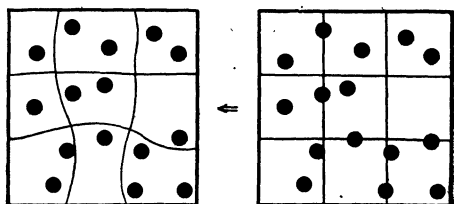


Рис. 6.

### 3.2. «НОВАЯ МАТЕМАТИКА»

Под «новой математикой» я понимаю некоторые попытки начальной школы имитировать формалистический подход профессиональных математиков. Я полагаю, что эта мода, одновременно охватившая многие школы в связи с новейшими тревогами общественности по поводу состояния начального обучения, плоха из-за свойственных ей разнообразных подмен содержания формой. Последние вызывают трудности и у учителя, и у ребенка.

Вследствие формального подхода учительница не сможет существенно помочь ребенку, у которого возникли затруднения с формулировками. Дело в том, что она будет чувствовать себя неуверенно, заставляя его заучить различие между пустым множеством и ничем или различие между «символами»  $3+5$  и цифрой 8, которая является «общим именем» числа восемь, надеясь, что любознательное дитя не спросит, каково общее имя дроби  $8/1$ , которое, наверное, отличается от рационального числа  $8/1$ , и от частного  $8/1$ , и от «указанного деления»  $8/1$ , и от упорядоченной пары  $(8/1)$ . Педагог воздержится от обсуждения параллельных прямых, поскольку знает, что они обычно не пересекаются, но слышал, что могли бы пересечься, если продлить их достаточно далеко. Он побоится, как бы не произошло что-нибудь наподобие того, что случилось однажды в эксперименте у какого-то русского математика. Впрочем, хватит говорить о проблемах учителя. Рассмотрим теперь три вида протестов с позиции ребенка.

**Протесты развития.** Очень плохо, когда утверждают, что ребенок хранит свои знания в виде простой упорядоченной иерархии понятий. Чтобы находить то, что ему нужно, он должен обладать многосвязной сетью, чтобы иметь возможность испытывать несколько способов достижения каждой цели. Он может не справиться с приспособлением именно первого способа к условиям задачи. На этом этапе упор на «формальное дока-



зательство» является деструктивным, потому что знания, нужные для поиска доказательств и их понимания, гораздо более сложны (и менее полезны), чем знания, упоминаемые в уже найденных доказательствах. Сеть знаний, требующаяся для понимания геометрии, сплетена из примеров и явлений, а также наблюдений за сходствами и различиями между ними. В детстве не кажется очевидным, что такие сплетения упорядочены, подобно аксиомам и теоремам логической системы, или что ребенок мог бы воспользоваться такой конструкцией, если бы обладал ею. *После* того как явление понято, может иметь большое значение создание для него формальной системы для облегчения понимания более развитых предметов. Но даже в этом случае такая формальная система является всего лишь одной из многих возможных моделей; сочинители новой математики, по-видимому, путают свою аксиомно-теоремную модель с самой системой чисел. Я утверждаю, что в случае аксиом арифметики соответствующий формализм может принести больше вреда, чем пользы для понимания более развитых предметов.

Исторически используемый в новой математике «множественный» подход возник из попытки формалистов вывести интуитивно воспринимаемые свойства континуума из теории почти конечных множеств. Они отчасти преуспели в этом трюке (или в «хекерстве», как выразились бы некоторые программисты), но столь сложным образом, что у тех, кто следует этой модели, исчезает возможность серьезно говорить о вещественных числах, пока не будут усвоены понятия, изучаемые на втором курсе высшей школы. Постижение идей топологии при этом откладывается на еще более поздний срок. Однако дети в свои шесть лет уже обладают хорошо развитыми геометрическими и топологическими идеями, только они плохо умеют манипулировать абстрактными символами и определениями. Нам следовало бы строить преподавание, основываясь на сильных сторонах ребенка, вместо того чтобы оболванивать его, пытаясь заменить то, чем он владеет, на структуры, которыми он еще не в состоянии оперировать. Но это как раз в духе математиков, — наверняка худших в мире растолковывателей, — думать: «Вы можете чему-нибудь научить ребенка, только если пользуетесь достаточно точными определениями»; «Если мы с самого начала все правильно определим, у нас впоследствии не возникнет никаких затруднений». Мы не программируем на Фортране для пустой машины, хуже того — мы лезем внутрь плохо понимаемой нами большой системы, для которой характерно при ее естественном эвристическом поведении использование многократно и различными способами описанных символов.

**Интуитивные протесты.** В новой математике акцентируется

идея, что число может быть идентифицировано с классом эквивалентности всех множеств, которые могут быть поставлены во взаимно однозначное соответствие друг с другом. При этом рациональные числа определяются как классы эквивалентности пар целых, и прилагается уйма усилий, чтобы *предостеречь ребенка от идентификации рациональных чисел с частными или дробями*. Функции часто интерпретируются как множества, хотя в некоторых текстах представлены «функциональные машины» с поверхностным привкусом понятия алгоритма. Определение «переменной» — это еще одно дьявольское хитросплетение имен, значений, выражений, фраз, предложений, символов, «указанных операций» и тому подобного. (На самом деле при реальном решении проблем возникает так много различных видов данных, что настоящие математики обычно не занимаются их формальной классификацией, а используют для их объяснения контекст, присущий конкретной проблеме.) В погоне за этой формалистической навязчивой идеей учебный курс никогда не представляет сколько-нибудь связной картины реальных математических явлений, т. е. процессов дискретных или непрерывных; алгебры, а не только столь усердно пережевываемого синтаксиса ее нотации; а также геометрии. «Доказываемые» время от времени «теоремы» наподобие такой: «У числа  $x$  имеется только одно аддитивное обратное число, —  $x$ » столь приземлены и очевидны, что ни преподаватель, ни учащийся не понимает цели таких доказательств. «Официальное» доказательство состоит в прибавлении величины  $y$  к обеим частям равенства  $x + (-y) = 0$ , применении закона ассоциативности, затем закона коммутативности, затем закона  $y + (-y) = 0$  и в заключение аксиом равенства, чтобы показать, что значение  $y$  должно равняться  $x$ . Детский ум может легче воспринять более глубокие идеи: «В равенстве  $x + (-y) = 0$ , если  $y$  меньше  $x$ , имелся бы некий излишек в левой части, а если  $x$  меньше  $y$ , в левой части имелось бы отрицательное число, и поэтому  $x$  и  $y$  должны быть в точности равны». Ребенку не разрешают пользоваться этим видом упорядоченно-непрерывного мышления преимущественно потому, что «в нем используются более продвинутые знания», поэтому он якобы не дает «настоящего доказательства». Но в нужной ребенку сети идей эта связь имеет равный логический статус и заведомо большее эвристическое значение. Приведу еще один пример. Учащегося заставляют четко различать операцию, обратную сложению, и расстояние, отсчитываемое в обратном направлении, хотя, казалось бы, желательно слияние этих понятий.

**Вычислительные протесты.** Идея процедуры и сноровка, которая приходит с изучением тестирования, модификации и приспособливания процедур, могут перейти во многие другие виды

действий ребенка. Такие традиционные академические предметы, как алгебра и арифметика, сравнительно мало влияют на развитие, особенно когда они не подкрепляются интуитивной геометрией. (Вопрос о том, какие виды учебных знаний помогают в других занятиях, является фундаментальным для теории обучения: я снова подчеркиваю наше предположение, что идеи процедуры и отладки окажутся уникальными по своему проникновению в самые различные сферы деятельности.) Мы уже отмечали, что в алгебре понятие «переменной» переусложнено, но в вычислениях ребенок может легко увидеть в  $x+y+z$  описание процедуры (любой процедуры сложения!), причем « $x$ », « $y$ » и « $z$ » указывают на данные для процедуры. Функции легко воспринять как процедуры, но это трудно сделать, представляя их упорядоченными парами. Если вам нужен граф, то опишите машину, которая чертит граф; когда у вас есть граф, опишите машину, которая может читать его для отыскания значений функций. В обоих случаях вы имеете дело с легкими и полезными понятиями.

Давайте избежим культурной ловушки; теоретико-множественные «основы» математики в наши дни популярны у математиков потому, что именно на них эти математики муштровались и воспитывались (в колледже). Эти ученые, как правило, просто незнакомы с вычислениями или с семейством теорий Поста — Тьюринга — Маккалоха — Питса — Маккарти — Ньюэлла — Саймона..., которые будут гораздо важнее, когда ребенок вырастет. Вопреки мнению логиков и издателей, теория множеств не является *единственным* и истинным основанием математики; этот подход отменно хорош для исследования бесконечности, но малоприменителен для понимания вещественных чисел и совсем бесполезен для изучения арифметики, алгебры и геометрии.

Если суммировать мои возражения, суть их состоит в том, что новая математика делает упор на формализм и манипуляции символами вместо того, чтобы заниматься эвристическим и интуитивным содержанием существа дела. Предполагается, что ребенок научится решать проблемы, но мы не учим его тому, что знаем сами — ни об учебном предмете, ни о решении задач<sup>1)</sup>.

<sup>1)</sup> В принципиальном, хотя и излишне веселом разборе руководств по новой математике Фейнман [20] объясняет следствия проведения различия между вещью и ее представлением. «Окрасьте в красный цвет рисунок мяча», — гласит книга вместо того, чтобы сказать: «Окрасьте мяч в красный цвет». «Должны ли мы красить весь квадрат, в котором присутствует изображение мяча, или только часть внутри окружности мяча?» — спрашивает Фейнман. («Окрасить мячи в красный цвет» должно бы, по-видимому, писаться так: «Окрасить внутренности всех окружностей всех элементов множества мячей» или примерно в этом духе.)

В качестве примера того, как озабоченность формой (в данном случае аксиомами арифметики) может исказить впечатление от существа, изучим странное пристрастие настаивать на том, что сложение в конечном счете является операцией ровно над двумя величинами. В новой математике выражение  $a+b+c$  должно «на самом деле» быть либо  $(a+(b+c))$ , либо  $((a+b)+c)$ , а выражение  $a+b+c+d$  может обрести содержательный смысл только после нескольких применений закона ассоциативности. Это во многих отношениях является глупостью. Ребенок уже обладает хорошим интуитивным представлением о том, что значит собрать несколько множеств вместе; ведь одинаково легко смешать в кучу шарики пяти или двух цветов. Таким образом, сложение уже воспринимается как  $n$ -местная операция. Однако прислушаемся к книжной попытке доказать, что это не так:

Сложение... всегда выполняется над двумя числами. На первый взгляд это может показаться необоснованным, так как вы раньше часто складывали длинные строки цифр. Попробуйте проэкспериментировать на себе. Попробуйте одновременно сложить числа 7, 8, 3. Вне зависимости от того, каким образом вы стараетесь сделать это, вы вынуждены выбрать два числа, сложить их, а затем прибавить третье число к их сумме.

(Из учебника для девятого класса.)

Разве высота башни — это результат попарного суммирования высоты ее ступенек в определенном порядке? Получается ли длина или площадь предмета аналогичным образом из его частей? Зачем им понадобилось вводить свои множества и их взаимно однозначные соответствия, чтобы потом упустить суть дела? Очевидно, что они так часто это повторяли, что и в самом деле поверили, будто выбранные ими аксиомы для алгебры обладают неким особым качеством истины в последней инстанции!

Рассмотрим несколько важных и занятных идей, которым не уделяется особого внимания в средней школе. Сначала рассмотрим сумму  $1/2 + 1/4 + 1/8 \dots$  Интерпретируя ее как площадь, человек овладевает пленительными идеями перегруппировки, продемонстрированными на рис. 7. Научившись делить, ребе-

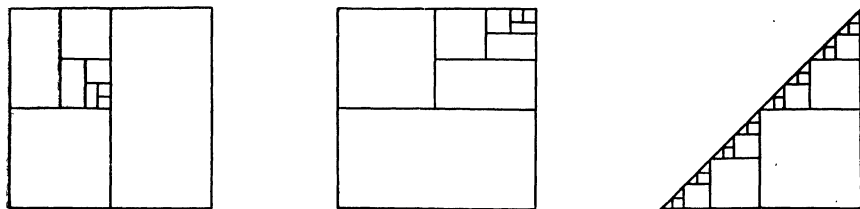


Рис. 7.

нок может вычислять и ощущать некоторые количественные аспекты предельного процесса  $0.5, 0.75, 0.875, 0.9375, 0.96875, \dots$  и может узнать о складывании и разрезании, об эпидемиях и популяциях. Он мог бы узнать о тождестве  $x = px + qx$ , где  $p + q = 1$ , и тем самым оценить процесс разведения, он может узнать, что  $3/4, 4/5, 5/6, 6/7, 7/8, \dots \Rightarrow 1$ , и начать понимать многие яркие и здравые геометрические и топологические следствия из таких отвлеченных материй.

Но в новой математике синтаксические границы между рациональными числами, частными и дробями заходят так далеко, что не разрешается для нахождения большей из дробей  $3/8$  и  $4/9$  вычислить и сравнить  $.375$  и  $.4444$ . От нас *требуют перекрестное перемножение* числителей и знаменателей. Хотя перекрестное перемножение очень привлекательно, ему присущи два дефекта: (1) никто не помнит, в какие стороны нужно переходить в зависимости от результата проверки условия, и (2) мы не получаем информации о том, насколько далеки друг от друга сравниваемые числа. Абстрактное понятие порядка весьма изящно (еще один набор аксиом для очевидного), но дети уже прекрасно понимают упорядоченность и хотят знать количественные соотношения.

Еще одной навязчивой идеей является понятие основания системы счисления. Хорошо, когда ребенок отчетливо понимает, что  $223$  — это «две сотни» плюс «двадцать» плюс «три», и я думаю, что эту мысль нужно довести до его сознания самым простым образом, а не усложнять ее<sup>1)</sup>. Я не считаю эту идею настолько богатой, что требуется муштровать малое дитя в арифметических упражнениях в нескольких системах счисления! Дело в том, что эта хилая идея мало связана с другими предметами, и существует риск изувечить хрупкую арифметику учеников, которые, усвоив с трудом, что  $6+7=13$ , теперь обнаруживают, что  $6+7=15$ . Кроме того, я не вижу в учебниках моих детей при всем внимании к основаниям систем счисления хоть каких-нибудь нетривиальных выводов — понятий, которые могли бы оправдать это внимание, например:

Почему десятичное целое число можно написать только одним способом?  
Почему работает отбрасывание девяток? (Это даже не упоминается.)  
Что произойдет, если пользоваться произвольными нестепенями, например,  $a+37b+24c+11d+\dots$  вместо обычных  $a+10b+100c+1000d+\dots$ ?

Если они не обсуждают таких вопросов, то должны иметь другую цель. Я предполагаю, что вся суeta поднята ради того, чтобы детишки лучше понимали процедуру умножения и деления. Но с точки зрения развития это может оказаться серьезной методической ошибкой для курсов обучения как старой,

<sup>1</sup> См. песню Т. Лейрера «Новая математика» [21].

так и «новой» математике. Стандартный алгоритм деления на бумаге является, мягко выражаясь, громоздким, и большинство детей никогда не воспользуются им для изучения числовых явлений. И хотя довольно любопытно понять, как работает этот алгоритм, выписывание всей картинки заставляет предположить, что педагог верит, будто ребенок должен всякий раз понимать черт знает что! Это неверно. Существенная идея алгоритма, если такая есть, состоит в повторяемом вычитании; все остальное является хитроумным, но не жизненно необходимым программистским трюком.

Если мы хотим научить, хотя бы методом зубрежки, практическому алгоритму деления, очень мило. Но в любом случае давайте снабдим детей маленькими калькуляторами; если это потребует слишком больших расходов, почему бы не отбавить правила? Но, пожалуйста, без невыносимых объяснений. Важно справляться с вещественными числами. Принятая в новой математике озабоченность целыми числами столь фанатична, что напоминает (если позволительно упомянуть другую псевдонауку) нумерологию. (Какого мнения об этом *Boston Globe*?)

Теоретико-множественный формализм Дедекинда — Рассела — Уайтхеда достойно дополнил следующую (после Эвклида) серию демонстраций того, что из немногих примитивов можно вывести много математических понятий, хотя бы и длинным и извилистым путем. Однако у ребенка проблема состоит вовсе не в том, чтобы овладеть хоть какими-то понятиями; ему нужно познавать реальный мир. С точки зрения доступных ему понятий весь формализм теории множеств «в подметки не годится» одной-единственной, более древней, более простой и, возможно, более великой идее: представления интуитивной вещественной прямой бесконечной десятичной дробью.

*Существует реальное противоречие между целями логика и преподавателя. Логик хочет минимизировать разнообразие идей, и его не беспокоит длинный, тонкий путь вывода. Педагог (в идеале) хочет сделать пути как можно короче, и его не смущают, а на самом деле даже радуют связи со многими другими идеями. И он почти вовсе не беспокоится о направлениях связей.*

Что же касается лучшего понимания целых чисел, то я полагаю, что в этом не могут помочь бесконечные упражнения малых детей, понуждаемых чертить схемы взаимно однозначного соответствия. Несомненно, эти упражнения помогут их обучению существенным алгоритмом, но не для чисел, а для важных топологических и процедурных проблем черчения путей без пересечений и тому подобного. Именно этой отнюдь не мало-важной проблеме нам следует уделять внимание.

Итак, специалист по информатике несет ответственность за образование. Не потому, как он ошибочно полагает, что он должен будет программировать обучающие машины. И, конечно, не потому, что он искусный пользователь «дискретной математики». Он знает, почему плохо пользоваться троюками, которые потом мстят за себя при отладке и расширении программ. (Так, можно привлечь интерес детишек, связав малые числа с произвольными цветами. Но как пригодится этот фокус в их дальнейших попытках применять понятие числа к площади, объему или к значению функции?) Именно специалист по информатике должен исследовать эти вопросы, потому что он владеет понятием процедуры — секретом, который так долго искали педагоги.

### ЛИТЕРАТУРА

1. Feynman R. P. Development of the space-time view of quantum electrodynamics. Science 153, No. 3537 (Aug. 196), 699—707.
2. Shannon C. E. A universal Turing machine with two internal states. In Automata Studies, Shannon C. E., and McCarthy J. ; (Eds.), Princeton U. Press, Princeton, N. J., 1956, p. 157—165.
3. Cook S. A. On the minimum computation time for multiplication. Doctoral diss., Harvard U., Cambridge, Mass., 1966.
4. Knuth D. The Art of Computer Programming, Vol. II. Addison-Wesley, Reading, Mass., 1969.
5. Minsky M. and Papert S. Perceptions: An Introduction to Computational Geometry. MIT Press, Cambridge, Mass., 1969.
6. Guzman A. and McIntosh H. V. CONVERT. Comm. ACM 9, 8 (Aug. 1966), 604—615.
7. Hewitt C. PLANNER: A language for proving theorems in robots. In: Proc. of the International Joint Conference on Artificial Intelligence, May 7—9, 1969, Washington, D. C., Walker, D. E. and Norton, L. M. (Eds.), pp. 295—301.
8. Levin M., et al. The LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1965.
9. Weissman C. The LISP 1.5 Primer. Dickenson Pub. Co., Belmont, Calif., 1967.
10. Martin W. A. Symbolic mathematical laboratory. Doctoral diss., MIT, Cambridge, Mass., Jan. 1967.
11. Moses J. Symbolic integration. Doctoral diss., MIT, Cambridge, Mass., Dec. 1967.
12. Seagle J. R. A heuristic program that solves symbolic integration problems in Freshman calculus. In Computers and Thought, Feigenbaum E. A. and Feldman J. (Eds.), McGraw-Hill, New York, 1963.
13. Turing A. M. Computing machinery and intelligence. Mind 59 (Oct. 1950), 433—460; перепечатка в Computers and Thought, Feigenbaum E. A. and Feldman J. (Eds.), McGraw-Hill, New York, 1963.
14. Minsky M. Matter, mind and models. Proc. IFIP Congress 65, Vol. 1, pp. 45—49 (Spartan Books, Washington D. S.). Перепечатка в Semantic Information Processing, Minsky M. (Ed.), MIT Press, Cambridge, Mass., 1968, pp. 425—433.
15. Feigenbaum E. A., and Feldman J. Computers and Thought. McGraw-Hill, New York, 1963.

16. Minsky M. (Ed.). *Semantic Information Processing*. MIT Press, Cambridge, Mass., 1968.
17. Galton F. *Inquiries into Human Faculty and Development*. Macmillan, New York, 1983.
18. Attneave F. Triangles as ambiguous figures. *Amer. J. Psychol.* 81, 3 (Sept. 1968), 447—453.
19. Papert S. Principes analogues a la recurrence. In *Problems de la Construction du Nombre*, Presses Universitaires de France, Paris, 1960.
20. Feynman R. P. New textbooks for the "new" mathematics. *Engineering and Science* 28. 6 (March 1965), 6—15 (California Inst. of Technology, Pasadena).
21. Lehrer T. New mat. In *That Was the Year that Was*, Reprise 6179, Warner Bros. Records.



# 1970

## Некоторые замечания математика-вычислителя

*Дж. Уилкинсон*

Национальная физическая лаборатория  
Великобритания, Мидлсекс, Теддингтон

### ВВЕДЕНИЕ

Когда я наконец оправился от радостного шока в связи с приглашением прочесть лекцию лауреата премии Тьюринга за 1970 г., то осознал, что и в самом деле должен подготовить приличествующую случаю лекцию. По-видимому, сложилась традиция, согласно которой лауреат должен сам решить, чего от него ждут, и, возможно, по этой причине предыдущие лекции заметно различались по стилю и содержанию. Однако мне было совершенно четко сказано, что я должен произнести послеобеденную речь и что под рукой у меня не будет ни проектора, ни доски.

Хотя я был связан с быстродействующими компьютерами со времени их появления, чести, которой я удостоился, я обязан главным образом своей работе в качестве математика-вычислителя, в особенности в области анализа погрешностей. Изучение программы этой встречи обнаружило, что как раз численный анализ здесь почти не представлен, и соответственно я почувствовал, что будет неуместно готовить серьезную лекцию о погрешностях округления; я думаю, впрочем, что вряд ли эта тема подойдет для послеобеденной речи в любой компании. Вследствие этого я решил сделать некоторые довольно личные замечания, основанные на моей практике вычислителя за последние двадцать пять лет.

Вполне вероятно, что в одном важном отношении я занимаю особое место среди лекторов-лауреатов.

Морис Уилкс, прочитавший лекцию лауреата премии Тьюринга в 1967 г., заметил, что вряд ли многие из последующих лауреатов будут людьми, лично знакомыми с Аланом Тьюрингом. Я же могу похвастаться гораздо большим. С 1946 до 1948 г. я имел счастье работать вместе с этим великим человеком в Национальной физической лаборатории. Я намеренно говорю «великий человек», потому что он и в самом деле был выдающийся гений. Для тех сотрудников нашей лаборатории, кто знал его и работал с ним, было очень радостно, что АСМ признала его огромные заслуги перед вычислительной наукой, учредив эту премию Тьюринга, а мне было особенно приятно

стать ее лауреатом вследствие моей связи с его работой во время важного периода его деятельности. Я надеюсь, что в таких обстоятельствах будет вполне уместно, если я посвящу часть моей речи периоду совместной работы с ним. Эта связь, несомненно, сильно повлияла на мою карьеру, и не будь ее, едва ли я продолжал бы заниматься вычислительной математикой.

## ВМЕСТЕ С АЛАНом ТЬЮРИНГОМ

По наклонностям и образованию я был чистый математик. В Кембридже 30-х годов еще доминировал классический анализ, и я был под сильным влиянием традиции Харди — Литтлвуда. Если бы не вторая мировая война, я почти наверное получил бы свою докторскую степень в этой области. Однако, достигнув призывного возраста, когда разразилась война, и будучи по природе патриотом, я увидел, что могу служить своей стране более эффективно и, кстати, в намного лучших условиях, работая в правительственной Научной службе, чем в качестве некомпетентного пехотинца. Английское правительство заняло удивительно просвещенную позицию по этому вопросу, и почти с самого начала войны люди с научной квалификацией поощрялись именно к такому образу действий.

Вследствие этого во время войны я служил в Департаменте военных исследований (который тесно взаимодействовал с Абердинским испытательным полигоном), работая главным образом над такими «очаровательными» темами, как внешняя баллистика, разброс осколков бомб и снарядов и термодинамика взрывчатых веществ. Моей задачей было решать проблемы математического характера, возникающие в этих областях, используя, если нужно, вычислительные методы. Всякий, кто когда-нибудь занимался такими вопросами, поймет, что они могут быть довольно суровым испытанием, и успехи здесь зачастую перемежаются неудачами. Вначале я не находил эту работу особенно близкой мне по духу, но постепенно увлекся численным решением физических задач. Позднее в этой лекции я опишу свое раннее знакомство с матричными вычислениями, которое значительно повлияло на мою последующую карьеру.

В конце войны не удалось сразу добиться увольнения, и в 1946 г. я перешел в только что образованное Отделение математики Национальной физической лаборатории. Именно там я впервые встретил Алана Тьюринга, хотя, конечно же, был наслышан о нем и прежде, правда, главным образом как об эксцентричном человеке. Теперь, когда информатика включает в себя по существу целых два отделения этой лаборатории и распространяет свои щупальца в остальные, интересно

напомнить, что в то время штат сектора высокоскоростного вычисления составлял 1,5 человека. Единицей был, конечно, никто иной, как сам Алан Тьюринг, а половиной был я. Спешу добавить, что с моей стороны это вовсе не ложная скромность. Я должен был проводить половину своего времени в вычислительном секторе, который находился в умелых руках Чарльза Гудвина и Лесли Фокса, и другую половину с Аланом Тьюрингом. В течение нескольких месяцев Алан и я работали вместе на чердаке старого дома в замечательно маленькой комнатке, которая была «временно» нанята лабораторией, чтобы приютить Отделение математики. Незачем говорить, что сейчас, двадцать пять лет спустя, эта комнатка все еще принадлежит лаборатории. Тьюринг так никогда и не стал строителем империи, он собирал свой штат довольно медленно и работал в тесном контакте с ним. Год спустя численность персонала сектора выросла лишь до 3,5; эти два новых сотрудника были Майк Вуджер, который более всего известен своей работой по Алголу, и Гарри Хаски (который не нуждается в представлении аудитории АСМ), проводивший 1947 г. в Национальной физической лаборатории.

Моей задачей было помогать Тьюрингу в логическом конструировании компьютера АСЕ, который должна была построить лаборатория, и рассмотреть проблемы программирования некоторых основных алгоритмов численного анализа, так что моя работа в вычислительном секторе предназначалась для расширения познаний в этой области. (Тем из вас, что знаком с работами Тьюринга, будет интересно узнать, что он называл последовательность команд, нужных для определенной задачи, соответствующей «таблицей команд» (instruction table) — термин, который повсюду вводил людей в заблуждение.) Легко представить себе, что такая работа не оставляла мне свободного времени. Работать с Тьюрингом было потрясающе интересно, и порой я доходил до полного истощения. Он только что сам стал горячо интересоваться проблемами численного анализа, и ему доставляло большое удовольствие подвергать Лесли Фокса, который был самым опытным специалистом по численному анализу в Национальной физической лаборатории, острой, но полезной критике методов, используемых последним.

Было невозможно работать «половину времени» с таким человеком, как Тьюринг, и почти с самого начала мои визиты в вычислительный сектор были довольно краткими. Однако двойное назначение имело свои полезные аспекты. У Тьюринга случались дни, когда он был «нелюдимым», и в такие дни рекомендовалось проявлять осторожность. Я скоро научился распознавать симптомы и использовал свое право (или, как обычно я представлял это, «выполнял свою обязанность») ра-

ботать в вычислительном секторе, пока не проходило его дурное настроение, что обычно бывало очень скоро.

Тьюринг имел обыкновение разрабатывать тему, начиная с исходных принципов, на первых порах обычно не заглядывая ни в какую уже существующую работу по данному предмету, и, несомненно, именно эта привычка придает его трудам столь характерный оригинальный почерк. Мне вспоминается замечание, которое, как говорят, сделал Бетховен, когда его спросили, слышал ли он одну вещь Моцарта, привлекающую общее внимание. Он ответил, что нет, и добавил: «И не буду слушать, чтобы не потерять что-то из моей собственной оригинальности».

Тьюринг довел этот принцип до крайних пределов, и должен сознаться, что поначалу это меня сильно раздражало. Он отводил мне часть работы и, когда я заканчивал ее, не удостоивал меня чести поглядеть на мое решение, но сам влезал в задачу; только после такого предварительного испытания он был готов прочесть то, что я сделал. Скоро я увидел преимущества такого подхода. Во-первых, он действительно не так быстро схватывал идеи других людей, как формулировал свои собственные, но, что более важно, часто находил какой-нибудь оригинальный подход, который ускользнул от меня и, возможно, не был бы замечен и им самим, прочитай он сразу мой отчет. Когда он наконец переходил к чтению моей работы, то обычно бывал великодушен в оценке; особенно он любил маленькие программистские трюки (кое-кто сказал бы, что он слишком их любил, чтобы быть «хорошим» программистом) и добродушно посмеивался, как мальчик, над маленькими хитростями, которые я использовал.

Когда я пришел в Национальную физическую лабораторию, я вовсе не думал оставаться там надолго и по-прежнему хотел вернуться в Кембридж, чтобы заняться исследованиями в области классического анализа. Период совместной работы с Тьюрингом зажег меня таким энтузиазмом в отношении проекта компьютера и так усилил мой интерес к численному анализу, что постепенно я оставил эту идею. Как я люблю выражаться в разговоре с моими друзьями из числа чистых математиков, «если бы не Тьюринг, я скорее всего стал бы всего лишь чистым математиком», стараясь придать этому замечанию достаточно пренебрежительный оттенок.

Репутация Тьюринга теперь так прочно установлена, что вряд ли нуждается в моей поддержке. Однако я должен сказать, что опубликованные им работы не дают полноценного представления о его замечательной разносторонности как математика. Его знания охватывали всю область чистой и прикладной математики и казались неотъемлемой частью самого человека, а не просто чем-то вычитанным из книг. Трудно было

себе представить, что он может что-то «забыть». Несмотря на это, он опубликовал лишь двадцать статей (и то если считать практически все), написанные за период около двадцати лет. Как ни замечательны некоторые из этих статей, все они представляют лишь малую долю того, что он, вероятно, сделал бы, сложись обстоятельства хоть чуть-чуть иначе.

Прежде всего шесть лет начиная с 1939 г. он провел в Форин Оффис. Ему было 27 лет в 1939 г., так что в других условиях этот период вполне мог быть самым продуктивным в его жизни. Но, кажется, он не жалел о времени, проведенном там, и у нас сложилось впечатление, что то был один из счастливейших периодов его жизни. Тьюринг просто любил задачи и головоломки всякого рода, и проблемы, с которыми он там встречался, видимо, были хорошей забавой для его ума. Конечно, именно там он приобрел свои познания в электронике, и это, вероятно, было решающим фактором в его решении пойти работать в Национальную физическую лабораторию, чтобы конструировать компьютер, а не вернуться в Кембридж. Математики склонны считать этот период «потерянными годами», но я думаю, что он был слишком разносторонним ученым, чтобы мыслить такими категориями.

Второй фактор, ограничивший его научную продуктивность, — явная неприязнь к перу и бумаге. Говорят, что уроки языка и литературы в школе вызывали у него мало энтузиазма, и, видимо, публикацию статей он находил еще более скучным и утомительным занятием, чем большинство из нас. Сам я считаю стиль его статей свежим и полным маленьких индивидуальных черточек, особенно дорогих для любого, кто его знал. В муках сочинительства он яростно стучал по клавишам старой пишущей машинки (он владел ею посредственно, чтобы не сказать сильнее), и именно в таких случаях было особенно разумно отправляться в вычислительный сектор.

Когда я готовил эту лекцию, был обнаружен старый отчет Отделения математики. Он написан Тьюрингом в 1946 г. для администрации лаборатории, и его главная цель — убедить начальство в возможности и важности построения электронного компьютера. Он полон характерных блесков юмора, и, перечитав этот документ впервые после 24 лет, я был снова поражен оригинальностью и многосторонностью его автора. Может быть, полезно напомнить, что еще в 1946 г. Тьюринг рассмотрел возможность работы как с интервальной арифметикой, так и с арифметикой значащих разрядов, и отчет напомнил забытые разговоры, не говоря уже о жарких спорах, которые мы вели по этому поводу.

Международная репутация Тьюринга основывается главным образом на его работах о вычислимых числах, но я хотел

бы напомнить, что он был и выдающимся специалистом по численному анализу и значительную долю своего времени начиная с 1946 г. посвятил этой области, хотя в основном в связи с решением физических задач. Работая в Национальной физической лаборатории, он написал замечательную статью об анализе погрешностей матричных вычислений [1], и я еще вернусь позже к этой статье.

Последние месяцы своего пребывания в лаборатории Тьюринг во все возрастающей степени был неудовлетворен развитием проекта ACE. Он всегда думал о большой машине с 200 линиями задержки, хранящими порядка 6000 слов, и мне кажется, что это был слишком честолюбивый проект для возможностей нашей лаборатории (и большинства других подобных учреждений) в то время.

Посетив лабораторию, Гарри Хаски попытался добиться, чтобы началась работа над менее претенциозной машиной, основанной на идеях Тьюринга. Алан не мог заставить себя поддержать этот проект; в 1948 г. он покинул Национальную физическую лабораторию и присоединился к группе в Манчестерском университете. Когда он ушел, четверо старших сотрудников сектора ACE Отделения математики и только что образованного сектора электроники объединили усилия и совместно вели разработку компьютера PILOT ACE, для которого мы использовали некоторые свои идеи, выработанные вместе с Гарри Хаски; следующие два-три года мы все работали инженерами-электрониками. Я думаю, мы можем утверждать, что PILOT ACE оказалась вполне успешной разработкой, и так как Тьюринг никогда бы не позволил этому проекту тронуться с места, мы, в этом отношении по крайней мере, выиграли от его ухода, хотя Отделение математики уже никогда не было таким, как прежде. Работа с гением имеет свои преимущества и свои неудобства! Однако, раз уж машина оказалась удачной, Тьюринг не делал кислых мин и всегда был предельно щедр в оценках достигнутого.

## СОВРЕМЕННОЕ СОСТОЯНИЕ ЧИСЛЕННОГО АНАЛИЗА

Я хотел бы теперь перейти к главной теме моей лекции — современному состоянию численного анализа. Численный анализ уникален среди различных предметов, которые вместе составляют довольно плохо определенную область информатики. Я делаю довольно-таки вызывающее замечание, потому что мне было бы очень жалко, если бы численный анализ порвал все свои связи с информатикой, хотя признаю, что на мой взгляд несомненно оказала известное влияние моя работа по созданию электронных компьютеров в то незабываемое время,

когда все только начиналось. Тем не менее численный анализ явно отличается от других разделов информатики тем, что имеет долгую и славную историю. Только название ново (кажется, до 50-х годов его не применяли), и это по крайней мере роднит его с информатикой.

Некоторые любят проследживать его историю вплоть до вавилонян, и если рассматривать всякое разумно организованное вычисление как численный анализ, то я думаю, что это вполне оправданно. Несомненно, что многие гиганты математического мира, включая великих Ньютона и Гаусса, посвящали значительную часть своих исследований вычислительным задачам. В то время математик не мог проводить таким образом свое время, не обращая внимания на критику коллег.

Многие лидеры компьютерной революции стремились разработать инструмент, специально предназначенный для решения задач, возникающих в физике и инженерном деле. Это несомненно так в отношении двух гениев — фон Неймана и Тьюринга, который столь много сделал, чтобы привлечь подлинно даровитых людей в вычислительную математику в ее ранние дни. Отчет Тьюринга, о котором я уже упоминал, показывает совершенно ясно, что он рассматривал подобные приложения как главное оправдание того, чтобы пускаться в такое сравнительно дорогостоящее, даже по тем временам, предприятие. Многие светила и лидеры возникших тогда вычислительных обществ были первоначально специалистами по численному анализу, и редакционные коллеги новых журналов были вынуждены большей частью из их рядов.

Использование вычислительных машин привело к появлению массы новых проблем, связанных в большей или меньшей степени с «программированием», и очень скоро вокруг компьютера выросла целая область новых исследований. В блестящей статье о численном анализе [2] Филипп Дэвис использует термин «компьютерология», чтобы охватить эту многообразную деятельность; правда, из осторожности он приписывает термин анонимному дружественному критику. В этой беседе я вовсе не намереваюсь применять этот термин в бранном смысле; но это — полезное собирательное слово для всей информатики, отличной от численного анализа. Многие люди, первоначально намеревавшиеся решать задачи математической физики, внезапно обнаружили, что они временно захлестнуты проблемами компьютерологии; и мы все еще ждем с затаенным дыханием их эпохальных открытий, которые они, конечно, сделают, когда вернутся в отчий дом, исполненные высшей мудрости.

В отличие от численного анализа проблемы компьютерологии абсолютно новые. Вся область характеризуется неустанной

активностью и возбуждением, и постоянно возникают совершенно новые вопросы. Хотя, несомненно, ряд направлений этой новой деятельности обречен на недолгую жизнь, компьютерология должна играть очень важную роль в том, чтобы компьютеры использовались на полную мощьность. Я уверен, что для математиков-вычислителей полезно быть связанными с группой людей, в которых так много жизни и энтузиазма. Точно так же я уверен, что информатика выигрывает от того, что включает в себя такой предмет, как численный анализ, с его прочной основой прошлых достижений. Неизбежно, однако, численный анализ стал выглядеть несколько консервативным внутри информатики, и появляются признаки того, что математики-вычислители начинают терять веру в себя. Их чувство изоляции усугубляется нынешней тенденцией к абстракции при обучении математике, что делает отношения напряженными. Как по-иному могло все сложиться, если бы компьютерная революция произошла в 19-м столетии! В своей статье Дэвис замечает, что люди уже начинают спрашивать: «Не умер ли численный анализ?» Дэвис дает собственный ответ на этот вопрос, и я не собираюсь приводить его здесь. Во всяком случае, «численный анализ» может быть назван «истэблишментом» в информатике, а сейчас во всех сферах стало модно находить в истэблишменте трупное окоченение.

Есть и второй вопрос, который задают все чаще. Он принимает многие разные облики, но, может быть, лучше всего его выражает прямая фраза: «Что нового в численном анализе?» Этот вопрос неизменно облачают в такую форму, которая не оставляет сомнений, что сам спрашивающий ответил бы «ничего» или, более вероятно, одним из вульгарных двухсловных синонимов, которыми так богат английский язык. Эта критика напоминает мне о несколько сходной ситуации, которая имела место по отношению к функциональному анализу. Те, кто воспитан в старой традиции, склонны считать, что «в функциональном анализе нет ничего нового; старые результаты попросту одеваются в новые одежды». Правды в этом как раз хватает, чтобы утвердить критиков в их предрассудках.

На мой взгляд, подобная критика неявно основана на ложном сравнении. Конечно, в компьютерологии все ново; в этом одновременно и ее привлекательность, и ее слабость. Совсем недавно я узнал, что компьютеры революционизируют астрологию. Гороскопы, составленные компьютером! — этого, конечно, никогда не было, и я понимаю, что это очень хорошо вознаграждается. Но если серьезно, не следовало ожидать, что численный анализ будет поставлен с ног на голову за одно-два десятилетия только потому, что мы дали ему новое имя и наконец получили удовлетворительные инструменты для работы.



За последние 300 лет многие лучшие умы математического мира пробовали свои силы на задачах, которые мы пытаемся решить. Неудивительно, что скорость нашего прогресса не может вполне соответствовать бурному прогрессу, так характерному для компьютерологии.

## НЕКОТОРЫЕ ДОСТИЖЕНИЯ В ЧИСЛЕННОМ АНАЛИЗЕ

Если вы склонны думать, что я собираюсь оправдываться в том, что никакого реального прогресса не было достигнуто, то спешу уверить вас, что у меня не было такого намерения. Готовя эту лекцию, я составил краткое обозрение того, что было сделано с 1950 г., и нашел его удивительно впечатляющим. Следующие несколько минут я хотел бы уделить небольшому рассказу о достижениях в области, с которой я лучше всего знаком, — матричным вычислениям.

К счастью, у нас есть небольшая книга, написанная В. Н. Фаддеевой [3], содержащая краткое и точное описание ситуации, существовавшей в 1950 г. Значительная часть книги посвящена решению задачи на собственные значения, и едва ли хотя бы один из методов, обсуждавшихся там, используется сегодня. Более того, что касается неэрмитовых матриц, даже методы, предложенные на конференции по теории матриц в 1957 г. в Уэйне, с тех пор почти полностью вышли из употребления. Используя нынешний вариант QR-алгоритма, можно получить с хорошей точностью систему собственных векторов и собственных значений плотной матрицы порядка 100 примерно за минуту. Затем при желании можно выдать строгие оценки величины погрешностей округления как собственных значений, так и собственных векторов, найдя заодно еще более точное решение. На конференции 1957 г. в Уэйне мы и думать не могли о таких возможностях. Особенно привлекательная черта этого прогресса — то, что именно осознание роли численной устойчивости, возникшее из успехов анализа погрешностей, сыграло важную роль в создании новых алгоритмов.

Сравнимые успехи были достигнуты в развитии итерационных методов для решения разреженных линейных систем типа тех, что получаются из дифференциальных уравнений в частных производных; здесь алгоритмические достижения прогрессировали одновременно с углублением понимания свойств сходимости итерационных методов. Что касается плотных систем, то здесь развитие новых алгоритмов было менее значительным, но наше понимание устойчивости стандартных методов претерпело полную трансформацию.

В этой связи я хотел бы сделать свои последние замечания о совместной работе с Тьюрингом. Когда в 1946 г. я поступил

на работу в Национальную физическую лабораторию, пессимистическое отношение к устойчивости методов исключения для решения линейных систем было в своем апогее и служило главной темой для разговоров. Были выведены оценки, должествующие показать, что погрешность решения будет пропорциональна  $4^n$ , и это означало, что бессмысленно решать системы даже довольно умеренного порядка. Я думаю, что будет правильно сказать, что в то время (1946 г.) именно наиболее выдающиеся математики были наиболее пессимистичны, тогда как менее даровитые, возможно, были не в состоянии оценить всей серьезности трудностей. Я не хочу указывать свое место на этой шкале, но я оказался в довольно-таки двусмысленном положении по следующей причине.

Случилось так, что еще в Департаменте военных исследований я столкнулся с матричными вычислениями, которые порядочно меня озадачили. После ряда неудач мне дали для решения систему двенадцати линейных уравнений. Я был очень рад, что наконец-то получил задачу, о которой «все знаю», и отправился со своим заданием, уверенный, что очень скоро вернусь с решением. Однако, когда я пришел в свою комнату, моя уверенность быстро испарилась. Множество из 144 коэффициентов вдруг стало казаться гораздо большим, чем когда я получал его. Я проконсультировался с несколькими книгами, бывшими под рукой; одна из них, между прочим, рекомендовала прямое применение правила Крамера, использующего определители! Потребовалось немного времени, чтобы понять, что это было не очень-то хорошее предложение, и я в конце концов решил выбрать модификацию гауссова исключения, которая сейчас называется схемой частичного выбора главного элемента.

Опасения относительно погрешностей округления для методов исключения в то время еще не поднимали голову, и я стал вычислять с десятью знаками больше из осторожности, чем потому, что ожидал каких-либо серьезных эффектов неустойчивости. Система была слабо плохо обусловленной, хотя в то время мы еще не так свободно использовали подобные бранные термины, и начав с коэффициента порядка единицы, я постепенно терял значащие цифры, пока, наконец, не привел последнее уравнение к виду, скажем, такому:

$$.0000376235 \ x_{12} = .0000216312.$$

Помню, в этом месте я подумал, что вычисленное  $x_{12}$ , полученное из такого соотношения, едва ли имеет больше шести верных знаков, даже если предположить, что не было накопления погрешностей округления, и я было решил вычислять ответы только с шестью знаками. Однако, как хорошо знают те из

Вас, кто работал с настольными электрическими арифмометрами, обычно делаешь меньше промахов, если твердо держишься установившихся приемов работы; и соответственно я вычислил все неизвестные с десятью знаками, хотя и был вполне уверен в абсурдности этого. Получилось так, что все компоненты решения были порядка единицы, чего и следовало ожидать, исходя из природы физической задачи.

Тогда, будучи к тому времени хорошо обученным вычислителем, я подставил мое решение в исходные уравнения, чтобы посмотреть, насколько они удовлетворяются. Так как  $x_1$  было получено из первого исходного уравнения, я начал с подстановки в 12-е уравнение. Примите во внимание, что в настольной машине скалярное произведение накапливается точно, давая в результате 20 значащих цифр. (Интересно, что в наше время мы почти всегда довольствуемся меньшим в отношении арифметических устройств компьютеров!) К моему изумлению, левая часть совпадала с заданной правой в десяти знаках, т. е. во всех знаках правой части! Это, сказал я себе, совпадение. Но, хотя и не очень быстро, последовало еще одиннадцать «совпадений». Я был в полном недоумении. Я был уверен, что ни одно из неизвестных не может иметь больше шести верных знаков, и все же согласие было столь хорошим, как если бы мне дали точный ответ и затем я округлил его до десяти знаков. Однако войну еще предстояло выиграть, и не было времени углубляться в размышления по поводу погрешностей округления; во всяком случае, я уже потратил в несколько раз больше времени, чем моя первоначальная самонадеянная оценка. Вследствие этого мой начальник уже далеко не так высоко оценил мою работу, но и он вынужден был признать, что поражен, когда я заявил, что имею «точное решение», соответствующее правой части, которая отличается от заданной лишь в десятом знаке.

Легко представить себе, что этот опыт живо всплыл у меня в памяти, когда я пришел в Национальную физическую лабораторию и встретил там озабоченность неустойчивостью методов исключения. Конечно, я все еще был уверен, что вычисленные мной ответы имеют самое большее шесть верных знаков, но озадачивало то, что в моей единственной встрече с линейными системами требовала объяснения именно удивительная *точность* решений (по крайней мере в смысле малых невязок). В тогдашней атмосфере лаборатории я решил не рисковать прослыть глупцом, выпячивая этот опыт.

Однако случилось так, что спустя некоторое время после моего прихода в Отделение математики поступила система из 18 уравнений, и после некоторого ее обсуждения мы решили оставить теоретизирование и начать вычисления. Система из

18 уравнений выглядит удивительно внушительно, даже если имеешь в прошлом опыт работы с двенадцатью, и поэтому мы договорились о совместной работе. Операцию проводили Фокс, Гудвин, Тьюринг и я, и мы выбрали гауссово исключение с полным выбором главного элемента. Тьюринг не проявлял особого энтузиазма отчасти потому, что не имел опыта работы с настольным калькулятором, и отчасти потому, что был убежден в провале дела. История повторилась замечательно точно. Снова система была слабо плохо обусловленной, последнее уравнение имело коэффициент порядка  $10^{-4}$  (в то время как исходные коэффициенты были порядка единицы), и невязки опять-таки были порядка  $10^{-10}$ , т. е. величины, соответствующей точному решению, округленному до десяти знаков. Интересно, что в связи с этим примером мы впоследствии выполнили один-два шага процесса, который теперь называется «итерационное уточнение», и это убедило нас, что первое решение имело примерно шесть верных знаков.

Я полагаю, что это нужно рассматривать как поражение для Тьюринга, который в то время более, чем кто-либо из нас, был привержен пессимистической школе. Однако я уверен, что этот эпизод произвел сильное впечатление на него и заставил его заново задуматься над проблемой погрешностей округления в методах исключения. Примерно год спустя он написал свою знаменитую статью «Погрешности округления в матричных процессах» [1], которая вместе со статьей Дж. фон Неймана и Голдстейна [4] в значительной степени рассеяла мрак в этой области. Второй раунд несомненно остался за Тьюрингом!

Этот анекдот, как мне кажется, довольно хорошо иллюстрирует неопределенность и путаницу, существовавшие в то время и разделявшиеся даже наиболее выдающимися людьми, работавшими в нашей области. По контрасту с этим, я думаю, мы можем с полным основанием заявить сегодня, что имеем достаточно полное понимание проблем устойчивости матричных вычислений не только в решении линейных систем, но также и в гораздо более трудной задаче о собственных значениях.

## НЕДОСТАТКИ МАТРИЧНЫХ ВЫЧИСЛЕНИЙ

Хотя мы можем считать успешными разработки алгоритмов для матриц и понимание их вычислительных особенностей, в некоторых других отношениях даже в этой области особых успехов не было. Наиболее важное упущение — недостаточная связь теории и практики. Использование алгоритмов и понимание проблемы устойчивости пользователями намного больше отстает от передовых разработок, чем это должно быть.

Основные проблемы матричных вычислений имеют преимущество простых формулировок, и мне кажется, что подготовка хорошо проверенных и хорошо документированных алгоритмов должна происходить немедленно после их разработки и анализа. Есть две причины, почему это не делается. Во-первых, подготовить полную документацию оказалось гораздо более трудной задачей, чем думалось раньше, и, во-вторых, этому не уделялось достаточно внимания. В последние два года появляются признаки того, что эти недостатки, наконец, преодолеваются работой над книгой «Справочник по автоматизированным вычислениям» [5] (эта работа в части, относящейся к матричным алгоритмам, концентрируется в Национальной лаборатории в Аргоне) и более общим проектом Bell Telephone Laboratories [6]. Я думаю, что имеет первостепенное значение, чтобы плоды всех усилий, затраченных на создание приемлемых алгоритмов, были полностью доступны людям, нуждающимся в них. Я пошел бы еще дальше и сказал бы, что достижение этого является нашим долгом перед обществом.

Вторая тревожная особенность работ в области матричных алгоритмов состоит в тенденции к изоляции от очень близких разделов. Я хотел бы, в частности, упомянуть линейное программирование и статистические вычисления. Ученые, занимающиеся линейной алгеброй и линейным программированием, образовывали до недавнего времени почти полностью непересекающиеся научные сообщества, и это, конечно, нежелательно. Стандартные вычисления, нужные в практической статистике, предоставляют наиболее прямую возможность для применения основных матричных алгоритмов, и тем не менее сотрудничества удивительно мало. Совсем недавно я видел статью хорошо известного практического статистика о сингулярном разложении, которая, по крайней мере в своем варианте, не содержала никаких ссылок на работу Кахана и Голуба, создавших такой замечательный алгоритм для этой цели. Ясно, что здесь виноваты обе стороны, но я думаю, что обеспечить использование результатов своей работы в смежных областях — это прежде всего долг специалистов по матричным вычислениям и требует активных действий. Опять-таки есть признаки, что эта изоляция разрушается. В Станфорде профессор Данциг, один из создателей линейного программирования, сейчас сотрудничает с Отделением информатики, а в Великобритании готовятся планы встречи экспертов по матричным вычислениям и членов статистического общества. Исторические прецеденты часто играют важную роль в разрушении барьеров, и интересно, что сотрудничество между специалистами по численному решению дифференциальных уравнений в частных производных и матричной алгебре было всегда чрезвычайно тесным.

Третья печальная черта — это отсутствие должного влияния математиков-вычислителей на технические характеристики и программное обеспечение компьютеров. На заре компьютерной революции конструкторы вычислительных машин и математики работали в тесном контакте, и зачастую это были одни и те же люди. Сейчас, к сожалению, со стороны вычислителей преобладает тенденция отказываться от всякой ответственности за построение арифметических устройств и от воздействия на наиболее важные характеристики программного обеспечения. Часто говорят, что использование компьютеров для научной работы представляет небольшую часть рынка, и вычислители стали покорно принимать средства, «предназначенные» для других целей, и выжимать из них все, что можно. Я не уверен, что это неизбежно, и если бы было достаточно единства в выражении наших требований, нет причин, почему бы их нельзя было удовлетворить. В конце концов, одно из главных достоинств электронного компьютера с точки зрения вычислителя — это его способность «делать арифметику быстро». Почему же арифметика должна быть столь скверной? Даже и здесь есть обнадеживающие начинания. Специального упоминания заслуживает работа В. Кахана, и в сентябре прошлого года в ходе конференции на эту тему, организованной известным производителем компьютеров, он среди других имел возможность выразить свои взгляды.

## ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Я убежден, что математические вычисления должны играть важную роль в будущем и что их вклад будет полностью достоин ожиданий великих пионеров компьютерной революции. Наибольшая опасность для вычислителей в настоящий момент проистекает от отсутствия веры в себя, для чего нет реальных оснований. Я думаю, что характер исследований в численном анализе неизбежно существенно изменится в следующем десятилетии. В первых двух десятилетиях мы концентрировались на основных проблемах, тех, что, к примеру, возникают в линейной и нелинейной алгебре и теории приближений. В будущем ретроспективном взгляде эта деятельность будет оцениваться как предварительная заточка инструментов, которые нужны для настоящей работы. Для успеха в этом деле будет важно эффективнее, чем мы это делали до сих пор, привлекать специалистов из числа прикладных математиков и математических физиков. Во время своей недавней поездки в Советский Союз я был поражен тем фактом, что большая часть исследований в численном анализе проводится по существу специалистами по математической физике, которые взялись за ре-

шение своих задач численными методами. Эти люди широко представлены в Академии наук. Хотя я думаю, что нам на Западе нечего бояться сравнения достижений, все же мне кажется, что сам подход в Советском Союзе заметно здоровее.

В Великобритании есть признаки, что наш подход уже меняется. В Университете Данди должен быть проведен год численного анализа, в течение которого многие из наиболее выдающихся вычислителей мира посетят Великобританию. Совсем недавно в Королевском обществе состоялось обсуждение, посвященное численному анализу. Это едва ли можно было предположить года два назад. Я с нетерпением жду времени, когда вычислительная математика будет доминировать среди приложений и снова займет центральное место на встречах ACM.

#### ЛИТЕРАТУРА

1. Turing A. M. Rounding-off errors in matrix processes. *Quart. J. Mech.* 1 (1948), 287—308.
2. Davis P. J. Numerical analysis. In *The Mathematical Sciences: A Collection of Essays*. MIT Press, Cambridge, Mass., 1969.
3. Фаддеев Д. К., Фаддеева В. Н. Вычислительные методы линейной алгебры. — М.: Физматгиз, 1960.
4. Wilkinson J. H. *Handbook for Automatic Computation*, Vol. 2. Linear Algebra. Springer-Verlag, Berlin.
6. Gentleman W. M., Traub J. F. The Bell Laboratories numerical mathematics program library project. *Proc. ACM 23rd Nat. Conf.*, 1968, Brandon/Systems Press, Princeton, N. J., pp. 485—490.

# 1971

## Общность в системах искусственного интеллекта

*Джон Маккарти*

Станфордский университет

Лекция, прочитанная Джоном Маккарти по случаю получения премии Тьюринга в 1971 г., никогда не издавалась. В публикуемом постскриптуме к лекции, написанном в 1986 г., автор пытается сохранить особенности и дух оригинала, а также прокомментировать его в свете достижений последних пятнадцати лет.

### ПОСТСКРИПТУМ

Моя Тьюринговская лекция 1971 г. называлась «Общность в системах искусственного интеллекта». Название темы оказалось слишком амбициозным, так как я обнаружил, что в то время не мог удовлетворительно представить в письменном виде мои мысли по этому поводу. Было бы лучше сделать обзор предшествующих работ, а не пытаться предлагать что-то новое, но тогда это не входило в мои привычки.

Я благодарен Ассоциации вычислительных машин за предоставленную возможность попробовать еще раз. К несчастью для нашей области науки, но, может быть, к счастью для этой статьи, проблема общности в задачах искусственного интеллекта осталась почти столь же нерешенной, как и тогда, хотя сейчас у нас есть много новых идей, которых не было в 1971 г. Статья в большой степени основывается на этих идеях, но она ни в коей мере не является полным обзором существующих к 1986 г. подходов к достижению общности. Место, отведенное каждой из обсуждаемых идей, пропорционально скорее степени моего знакомства с этой идеей и не отвечает какому-либо объективному критерию.

В 1971 г. и даже еще в 1958 г. было очевидно, что системы искусственного интеллекта страдают отсутствием общности. Этот факт продолжает быть очевидным и в настоящее время, и эта очевидность обрастает все более детальными подробностями. Первый явный симптом состоит в том, что небольшое добавление к идее программы часто приводит к переписыванию заново всего начала со структурами данных. Некоторый прогресс был достигнут с появлением модульности структур данных, но по-прежнему невозможно, не переписывая программы, немного изменить стратегию поиска.



Другим симптомом является тот факт, что никто не знает, как создать базу данных, содержащую общепользные знания об окружающем мире (знания «здравого смысла»), которую могла бы использовать любая программа, нуждающаяся в этих знаниях. Вместе с другой информацией такая база данных содержала бы необходимые роботу знания о взаимодействии движущихся объектов в окружающей среде, знания о том, что человек, как правило, знает о своей семье, и факты, касающиеся купли-продажи. Это не зависит от того, выражены ли знания на логическом языке или с помощью какого-то другого формализма. Когда мы используем логический подход к искусственному интеллекту, недостаток общности проявляется в том, что аксиомы, которые мы создаем для выражения знаний «здравого смысла», оказываются слишком ограниченными, чтобы областью их применимости была вся общая база знаний. По моему мнению, выработка языка выражения общих знаний «здравого смысла» для включения их в универсальную базу данных — это ключ к достижению общности в системах искусственного интеллекта.

Здесь представлены некоторые идеи о том, как достигнуть общности, высказанные как до, так и после 1971 г. Я хотел бы еще раз повторить свое отречение от всеобъемлемости обзора.

## **ПРЕДСТАВЛЕНИЕ ПОВЕДЕНИЯ С ПОМОЩЬЮ ПРОГРАММ**

Фридберг [7, 8] изучал абсолютно общий подход к представлению поведения и предусмотрел способ обучения для его совершенствования. А именно, поведение представляется с помощью программы, а обучение происходит с помощью случайных модификаций программы и тестирования модифицированной программы. Подход Фридберга оправдал себя только при обучении тому, как перенести один бит информации из одной ячейки памяти в другую, а схема вознаграждения инструкций, вошедших в успешные проходы программы, путем уменьшения вероятности их модификаций, как было показано Саймоном [24], уступает способу тщательного тестирования каждой программы и отбрасыванию любой из них, если она оказалась неудовлетворительной. Похоже, никто не пытался следовать идее обучения при помощи изменения всей программы целиком.

Недостаток подхода Фридберга состоит в том, что, хотя представление поведения с помощью программ является вполне общим понятием, изменение поведения малыми модификациями программы сильно зависит от конкретной ситуации. Небольшие концептуальные изменения поведения обычно нельзя

представить с помощью малых изменений, внесенных в программу, особенно если это программа, написанная на машинном языке, и любая малая модификация текста программы одинаково вероятна.

Может быть, стоило бы попробовать что-нибудь более похожее на генетическую эволюцию, например дублировать подпрограммы, одни копии модифицировать, а другие оставлять неизменными. Обучающаяся система экспериментировала бы: а не лучше ли заменить некоторые вызовы начальных подпрограмм вызовами модифицированных подпрограмм? Скорее всего, этот подход также не будет работать, за исключением тех случаев, когда малые изменения поведения действительно можно получить с помощью небольших модификаций подпрограмм. Может быть, потребовалось бы зарезервировать для модификаций некоторое количество параметров подпрограмм.

Хотя Фридберг занимался задачей обучения на основании опыта, все способы представления знаний с помощью программ наталкиваются на аналогичные трудности, коль скоро придется комбинировать несопоставимые знания или создавать программы, модифицирующие знания.

## УНИВЕРСАЛЬНЫЙ РЕШАТЕЛЬ ЗАДАЧ И ЕГО НАСЛЕДНИК

Один из видов общности в искусственном интеллекте включает в себя методы нахождения решений, не зависящие от проблемной области. Аллен Ньюэлл, Херберт Саймон и их коллеги и студенты были первыми в разработке этого подхода и продолжают им заниматься.

Ньюэлл и его коллеги впервые представили универсальный решатель задач (GPS)<sup>1)</sup> на суд ученых в 1957 г. [18]. Исходной идеей являлось представление задач из некоего общего класса как задач преобразования одного выражения в другое при помощи множества допустимых правил. В работе [20] даже утверждалось, что развитие GPS может само мыслиться как задача этого класса. По моему мнению, GPS не мог быть действительно универсальным решателем задач, потому что в общем случае задачи нельзя представить в таком виде и потому что большинство знаний, необходимых при решении задач и достижении целей, не так-то просто представить в виде правил преобразования выражений. Но все же GPS был первой системой, в которой решающие задачу структуры целей и подцелей были отделены от конкретной области задач.

---

<sup>1)</sup> General Problem Solver.

Если бы оказалось, что универсальный решатель задач действительно универсален, то, возможно, были бы предсказания Ньюэлла и Саймона о быстром успешном развитии систем искусственного интеллекта. В настоящее время Ньюэлл предлагает для универсального представления задач использовать систему SOAR, которая, насколько я понимаю, работает с преобразованием одного состояния в другое, причем состояния не обязательно должны быть описаны с помощью выражений.

## ПРОДУКЦИОННЫЕ СИСТЕМЫ

Первые продукционные системы были созданы Ньюэллом и Саймоном в 1950-х гг., а их идея была впервые описана в работе [21]. Некоторая общность достигается использованием общего механизма целенаправленного поиска для всех типов задач, а изменяются только конкретные продукции. Ранние продукционные системы превратились в оболочки экспертных систем, широко распространенные в настоящее время.

Системы продукции представляют знания в виде фактов и правил, и почти всегда правила сильно синтаксически различаются между собой. Факты обычно соответствуют базисным примерам логических формул, т. е. символам предикатов, примененным к постоянным выражениям. В отличие от систем, основанных на логике, эти факты не содержат переменных или кванторов. Новые факты являются результатом вывода, наблюдения или вводятся пользователем. Переменные зарезервированы для правил, которые обычно имеют вид «образец — действие» (действия по эталону). Правила вводятся в систему программистом в области инженерии знаний и в большинстве систем не могут возникать в результате работы самой системы. В обмен на принятие этих ограничений создатель системы продукции получает относительно быстродействующую программу.

Системы продукции редко используют фундаментальные знания из тех областей, для которых они созданы. Например, в медицинской системе MYCIN [2] имеется много правил, как, исходя из симптомов и результатов лабораторных анализов определить, какая именно бактерия является причиной заболевания. Однако в ее формализме нет способа описать тот факт, что бактерии — это организмы, которые живут внутри тела человека. Кроме того, в системе MYCIN нет способа представления процессов, протекающих во времени, хотя в других системах продукции имеется возможность представлять процессы приблизительно на уровне ситуационного исчисления, которое будет описано в следующем разделе.

В системе продукций результатом сопоставления с эталоном является замена аргументов в эталонной части правила константами. Следовательно, система продукций не выводит общих суждений. Например, рассмотрим определение: контейнер стерилизован, если он запечатан и бактерии не могут в него попасть, а все бактерии, находящиеся внутри, мертвы. Система продукций (или логическая программа) может воспользоваться этим утверждением, только подставляя различные виды бактерий вместо переменных. Следовательно, она не может сделать вывод, что нагревание запечатанного контейнера приведет к его стерилизации, если известно, что бактерии погибают при нагревании, потому что она не может рассуждать о неопределенном множестве бактерий в контейнере. Дальнейшее обсуждение этих идей приведено в работе [14].

## ПРЕДСТАВЛЕНИЕ ЗНАНИЙ НА ЯЗЫКЕ ЛОГИКИ

В 1958 г. мне казалось, что малые модификации поведения в большинстве случаев можно описать как малые изменения представлений о мире, и для этого необходима система, точно отражающая эти представления.

Если вы хотите, чтобы машина могла выводить абстракции, скорее всего, это значит, что она должна уметь представлять эти абстракции некоторым достаточно простым способом [11, стр. 78].

В 1960 г. возникла идея увеличения общности, которая заключается в том, чтобы воспользоваться логикой для такого описания фактов, которое не зависело бы от того, как эти факты будут использоваться впоследствии. Тогда мне казалось (как, впрочем, и сейчас), что люди по объективным причинам предпочитают общаться с помощью декларативных предложений, а не языков программирования, все равно, является ли субъект общения человеком, существом с Альфа Центавра или компьютерной программой. Более того, и для внутреннего представления проявляются преимущества декларативной информации. Основным преимуществом декларативной информации является общность. Факт, что при столкновении двух объектов они производят шум, может быть использован в различных конкретных ситуациях: произвести шум, избежать шума, объяснить шум или объяснить отсутствие шума. (Я думаю, что те машины не столкнулись, потому что, услышав скрип тормозов, я не услышал звука от удара.)

При построении системы искусственного интеллекта с декларативным представлением информации нужно решить, какой тип декларативного языка использовать. В простейших системах допускается только применение постоянных предика-

тов к постоянным символам, например *on(Block1, Block2)*. Далее можно допустить применение любых термов, построенных из функциональных символов и символов предикатов, например *location(Block1)-top(Block2)*. Базы данных, написанные на Прологе, допускают произвольные хорновские выражения, содержащие и несвязанные переменные; например,  $P(x, y) \wedge Q(y, z) \supset R(x, z)$  — это прологовское выражение в обозначениях стандартной логики. Более высокий уровень — это логика первого порядка, в которую входят как кванторы существования, так и кванторы общности и произвольные формулы первого порядка. В рамках логики первого порядка выразительна сила языка зависит от того, каким классам объектов принадлежат переменные. Довольно значительную выразительную силу дает использование теории множеств, которая позволяет работать с выражениями для множеств любых объектов.

За любое увеличение выразительной силы приходится платить требуемой сложностью программ, осуществляющих рассуждения и решающих задачи. Другими словами, ограничение выразительности декларативной информации позволяет упростить процедуру поиска. Пролог представляет собой локальный оптимум в этом континууме, потому что хорновские выражения обладают средней выразительностью, но легко интерпретируются логическим решателем задач.

Одно из основных ограничений, которое обычно принимается, состоит в требовании, чтобы при выведении новых фактов таковыми являлись только формулы без переменных, т. е. в требовании производить рассуждения в высказываниях, подставляя вместо переменных константы. Оказывается, что повседневная жизнь человека по большей части сопровождается именно такими рассуждениями. В принципе Пролог идет дальше этого, потому что выражения, которые прологовские программы находят как значения переменных, могут сами содержать несвязанные переменные. Однако эта возможность редко используется, кроме как для получения промежуточных результатов.

Операция, которая невозможна без привлечения исчисления предикатов в большей степени, чем позволяет Пролог, — это универсальное обобщение. Рассмотрим логическое обоснование консервирования. Мы говорим, что контейнер стерилен, если он запечатан и все бактерии в нем погибли. Это можно написать в виде следующего фрагмента прологовской программы:

*sterile(X) :- sealed(X), not alive-bacterium(Y, X).*

*alive-bacterium(Y, X) :- in(Y, X), bacterium(Y), alive(Y).*

Однако прологовская программа, в которую входит этот фрагмент, может простерилизовать контейнер, только убивая персонально каждую бактерию, и при этом она будет требовать, чтобы какая-нибудь другая часть программы последовательно генерировала их имена. Ее нельзя использовать для того, чтобы логически обосновать процесс консервации: запечатывание контейнера и затем его нагревание, чтобы убить все бактерии сразу. Для рассуждений, приводящих к логическому обоснованию консервирования, необходимо использовать кванторы.

Я считаю, что программы, осуществляющие рассуждения и решение задач, должны в конечном счете допустить неограниченное использование кванторов и множеств и иметь достаточно строгие методы контроля, чтобы избежать комбинаторного взрыва.

Хотя идея, выдвинутая в 1958 г., была принята очень хорошо, в те годы было сделано очень мало попыток воплотить ее в программу. Основная из них — это диссертация Блэка на соискание степени доктора философии, защищенная в Гарварде в 1964 г. Сам я большую часть времени потратил на то, что считал предварительными проектами; главным образом это был LISP. Я хотел вначале понять, как знания здравого смысла можно выразить на языке логики, и это основная причина того, что я не стал сразу пытаться реализовать эту идею. Представление знаний здравого смысла на логическом языке и сейчас является моей целью. Мое стремление продолжать работу в этом направлении могло бы и поколебаться, если бы люди, развивающие нелогические подходы, достигли серьезных успехов в обеспечении общности.

Маккарти и Хэйес [12] стали различать эпистемологические и эвристические аспекты задач искусственного интеллекта и предположили, что изучать проблемы общности гораздо проще в рамках эпистемологического подхода. Разница состоит в том, что при эпистемологическом подходе требуется полный набор фактов, гарантирующий, что некоторая стратегия достигает цели, в то время как эвристический подход предполагает поиск приемлемой стратегии исходя из наличных фактов.

Идеей работы [11] было создание базы данных «здравого смысла» общего назначения. Информацию «здравого смысла», имеющуюся у людей, предполагалось записать в логической форме и включить в базу данных. Любая программа целенаправленного поиска могла бы обратиться к ней за фактами, необходимыми для того, чтобы решить, как достигнуть поставленной цели. Наиболее значимыми фактами базы данных должны были быть факты о результатах действий робота, пытающегося перемещать объекты с одного места на другое. Изу-

чение этой проблемы привело к созданию в 1960 г. *исчисления ситуаций* [12], целью которого было найти способ описания результатов действий (операций) вне зависимости от проблемной области.

Основной формализм ситуационного исчисления — это выражение вида

$$s' = \text{result}(e, s),$$

где  $s'$  — ситуация, возникающая, когда происходит событие  $e$  в ситуации  $s$ . Ниже приведены некоторые аксиомы ситуационного исчисления для перемещения и раскрашивания блоков.

Аксиомы результатов действий

- $\forall x l s. \text{clear}(\text{top}(x), s) \wedge \text{clear}(l, s) \wedge$   
 $\neg \text{tooheavy}(x) \supset \text{loc}(x, \text{result}(\text{move}(x, l), s)) = l.$   
 $\forall x c s. \text{color}(x, \text{result}(\text{paint}(x, c), s)) = c.$

Аксиомы фреймов

- $\forall x y l s. \text{color}(y, \text{result}(\text{move}(x, l), s)) = \text{color}(y, s).$   
 $\forall x y l s. y \neq x \supset \text{loc}(y, \text{result}(\text{move}(x, l), s)) = \text{loc}(y, s).$   
 $\forall x y c s. \text{loc}(x, \text{result}(\text{paint}(y, c), s)) = \text{loc}(x, c).$   
 $\forall x y c s. y \neq x \supset \text{color}(x, \text{result}(\text{paint}(y, c), s)) = \text{color}(x, s).$

Заметим, что во все описания исполнения действий посылки входят в явном виде и что утверждения (называемые аксиомами фреймов) о том, что не изменяется при выполнении того или иного действия, также выписаны в явном виде. Без этих утверждений было бы невозможно что-либо сказать о выражении  $\text{result}(e2, \text{result}(e1, s))$ , так как мы не знали бы, выполнены ли в выражении  $\text{result}(e1, s)$  посылки для того, чтобы событие  $e2$  имело ожидаемый результат.

Заметим также, что ситуационное вычисление применимо только в том случае, когда рассуждения о дискретных событиях, результатом каждого из которых является новая общая ситуация, имеют смысл. Непрерывные события и события, происходящие одновременно, теорией не охватываются.

Оказалось, что, к сожалению, практически невозможно использовать ситуационное исчисление предложенным выше способом даже для довольно ограниченных задач. Использование универсальных программ для доказательства теорем приводило к слишком медленной работе программы, потому что в 1969 г. программы для доказательства теорем [9] не имели средств управления поиском. Все это привело к созданию системы STRIPS [6], в которой используются только логические рас-

суждения в рамках конкретной ситуации. Формализм системы STRIPS был более ограниченным, чем исчисление ситуаций в полном объеме. Чтобы не ввести противоречия, необходимо было аккуратно выбирать факты, входившие в число аксиом. Эти противоречия могли возникать при невозможности удалить высказывание, которое не являлось бы истинным в результате происшедшей в результате действия ситуации.

## НЕМОНОТОННОСТЬ

Вторая проблема, связанная с аксиомами ситуационного исчисления, состоит в том, что они также не обладают достаточной общностью. Это *проблема уточнения*, и до конца 1970-х гг. не было найдено способа ее решения. Предположим, что мы хотим ввести в *базу данных «здравого смысла»* аксиому о том, что птицы могут летать. Очевидно, что аксиому нужно каким-то образом уточнить, потому что пингвины, мертвые птицы, а также птицы, ноги которых замурованы в бетон, летать не могут. Аккуратно сформулировать аксиому удастся, если ввести в нее исключения — пингвинов и мертвых птиц; однако ясно, что можно придумать сколько угодно дополнительных исключений, таких, как птицы, у которых ноги замурованы в бетон. Формализованные немонотонные рассуждения (см. [4], [15]—[17] и [23]) дают возможность сказать на формальном языке, что птицы умеют летать, если только ситуация не является ненормальной, а также сделать заключение, что будут рассматриваться только те ненормальные ситуации, появление которых является следствием принятых во внимание фактов.

Немонотонность значительно расширила возможности выражения универсальных знаний о результатах событий в ситуационном исчислении. Она также дала метод решения *проблемы фрейма*, которая была еще одним препятствием для достижения общности, как отмечено в [12]. Проблема фрейма (этот термин используется по-разному, но я первый его ввел) возникает в случаях, когда допустимыми являются несколько действий, каждое из которых изменяет определенные признаки ситуации. Необходимо каким-то образом сказать, что действие изменяет только те признаки ситуации, к которым оно непосредственно относится. Когда имеется фиксированное множество действий и признаков, можно прямо указать, какие признаки не изменяются в результате действия, даже если для этого потребуется много аксиом. Однако если предположить, что в базу данных можно вводить дополнительные признаки ситуаций и дополнительные действия, то возникнет проблема, суть которой в том, что аксиоматизация действий никогда не будет



завершена. Маккарти [16] придумал способ, как можно с этим справиться, используя методику ограничений (*circumscription*), но Лифшиц [10] показал, что этот метод надо улучшить, и сделал свои предложения по этому поводу.

Ниже приведены некоторые аксиомы ситуационного исчисления для перемещения и раскрашивания блоков с использованием метода ограничений (*circumscription*) из [16].

Аксиомы о расположении и результатах движения объектов

$$\forall x e s. \neg ab(aspect1(x, e, s)) \supset loc(x, result(e, s)) = loc(x, s).$$

$$\forall x l s. ab(aspect1(x, move(x, l), s)).$$

$$\forall x l s. \neg ab(aspect3(x, l, s)) \supset loc(x, result(move(x, l), s)) = l.$$

Аксиомы о цветах и раскрашивании

$$\forall x e s. \neg ab(aspect2(x, e, s)) \supset color(x, result(e, s)) = color(x, s).$$

$$\forall x c s. ab(aspect2(x, paint(x, c), s)).$$

$$\forall x c s. \neg ab(aspect4(x, c, s)) \supset color(x, result(paint(x, c), s)) = c.$$

Эти выражения показывают, как можно обойти проблему уточнения, потому что любое вообразимое количество условий, препятствующих перемещению или раскрашиванию, может быть добавлено позже, что повлечет за собой введение соответствующего *ab aspect...* Они также демонстрируют способ решения проблемы фрейма, при применении которого мы не должны декларировать, что перемещения не влияют на цвета и их расположение.

Даже после возникновения формализованных немонотонных рассуждений задача создания универсальной базы данных «здравого смысла» не поддается решению. Проблема заключается в том, чтобы написать аксиомы, которые удовлетворяли бы нашим замечаниям об объединении универсальных фактов о каком-либо явлении. Как только мы приняли предварительное решение о некоторых аксиомах, у нас уже есть средства, чтобы учесть ситуации, в которых они неприменимы, и сделать необходимые обобщения. Более того, предполагаемые трудности часто оказываются похожими на те, которые относились к птицам с замурованными в бетоне ногами.

## ВОПЛОЩЕНИЕ

Чтобы рассуждать о знаниях, представлениях или целях, необходимо расширить множество объектов, о которых ведутся рассуждения. Например, программа, которая осуществляет об-

ратный логический вывод о целевых установках, непосредственно использует их как предложения:  $on(Block\ 1, Block\ 2)$ ; это значит, что символ  $on$  используется как предикатный символ языка. Однако программе, которая хочет сказать, что выполнение  $on(Block\ 1, Block\ 2)$  должно быть отложено до того момента, как будет выполнено  $on(Block\ 2, Block\ 3)$ , необходимо предложение типа  $precedes (on(Block\ 2, Block\ 3), on(Block\ 1, Block\ 2))$ , и если оно является предложением в рамках логики первого порядка, то символ  $on$  должен восприниматься как функциональный, а выражение  $on(Block\ 1, Block\ 2)$  должно рассматриваться как объект в логическом языке первого порядка.

Такой процесс изготовления объектов из предложений или других категорий логического языка называется воплощением (*reification*). Этот процесс необходим для увеличения выразительной силы языка, но он опять-таки приводит к усложнению рассуждений. Обсуждение вопроса содержится в работе [13].

## ФОРМАЛИЗАЦИЯ ПОНЯТИЯ КОНТЕКСТА

По поводу каждой написанной аксиомы критически настроенный человек может сказать, что она верна только в определенном контексте. При наличии некоторой изобретательности критик может придумать более общий контекст, в котором аксиома в ее точном выражении неверна. Это особенно хорошо видно при рассмотрении того, как человеческое мышление отражается в языке. Попробуем аксиоматизировать предлог «на» («on»), для того чтобы потом сделать соответствующие выводы из информации, содержащейся в предложении «Книга лежит на столе» («The book is on the table»). Критик может затеять спор о точном значении слова «на» («on»), придумывая разнообразные трудности, состоящие в том, что что-нибудь может находиться между книгой и столом, или в определении необходимой для употребления слова «на» («on») величины гравитационной постоянной, а также следует ли при этом учитывать центробежные силы. Таким образом, мы сталкиваемся с сократовскими вопросами о том, что означают понятия, взятые в их абсолютной общности, и с примерами, которые никогда не возникают в реальной жизни. На самом деле наиболее общего контекста просто не существует.

Напротив, если аксиоматизация проведена на действительно высоком уровне обобщения, то аксиомы часто становятся длиннее, чем это необходимо для конкретных ситуаций. Поэтому люди предпочитают говорить «Книга лежит на столе» («The book is on the table»), пренебрегая указаниями на время

и определениями, на каком именно столе и какая именно книга. Проблема определения необходимого уровня общности обязательно возникает при выражении знаний «здорового смысла» на формальном языке, будь то логика, программа или какой-нибудь другой формализм. (Некоторые ученые считают, что на внутренних уровнях знания представляются только в виде примеров, а строгие механизмы применения аналогий и подобия позволяют их использовать на более общем уровне. Я желаю этим ученым удачи в формулировке точных утверждений о том, что же это за механизмы.)

Возможное решение проблемы состоит в формализации понятия контекста и в комбинировании его с методом ограничений (*circumscription*) в немонотонных рассуждениях. Добавим в функции и предикаты в наших аксиомах еще один параметр контекста. Каждая аксиома содержит утверждение о некотором контексте, в котором она справедлива. Следующие аксиомы говорят о том, что факты наследуются и в более ограниченном контексте, если не утверждаются исключения. Применимость каждого утверждения немонотонным образом распространяется на определенные более общие контексты, но здесь опять есть исключения. Например, в утверждении о том, что птицы летают, неявным образом подразумевается, что существует атмосфера (чтобы в ней летать). В более общем контексте это может и не предполагаться. Остается определить, чем наследование при переходе к более общему контексту отличается от наследования при переходе к частному контексту.

Предположим, что любое предложение  $p$ , находящееся в памяти компьютера, рассматривается в определенном контексте и является сокращенной записью выражения  $holds(p, C)$ , где  $C$  — название контекста. Некоторые контексты могут быть очень частными. Например, Ватсон — это доктор в контексте рассказов о Шерлоке Холмсе и психолог-баритон в трагической опере об истории психологии.

Отношение  $c1 \leq c2$  означает, что контекст  $c2$  является более общим, чем контекст  $c1$ . Разрешаются предложения вида  $holds(c1 \leq c2, c0)$ , так что предложения, относящиеся к контекстам, сами могут выполняться в определенном контексте. Теория не определяет «самый общий» контекст, так же как и теория множеств Цермело — Френкеля не определяет множество самого общего вида.

Логическая система, использующая контексты, может иметь операции входа в контекст и выхода из контекста, результатом применения которых является то, что можно было бы назвать *ультраестественным выводом*, допускающим последовательные рассуждения вида

$holds(p, C)$   
 $ENTER C$   
 $p$   
 $\vdots$   
 $\vdots$   
 $\vdots$   
 $q$   
 $LEAVE C$   
 $holds(q, C).$

Это напоминает обычные логические системы естественного вывода, но по причинам, выходящим за рамки данной статьи, может оказаться некорректным считать контексты эквивалентами множеств предположений и даже бесконечных множеств предположений.

Все это довольно неопределенно, но это гораздо больше, чем я мог сказать в 1971 г.

#### ЛИТЕРАТУРА

1. Black F. A deductive question answering system. Ph. D. dissertation, Harvard Univ., Cambridge, Mass., 1964.
2. Buchanan B. G., Shortliffe E. H., Eds. Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. American Elsevier, New York, 1984.
3. Davis R., Buchanan B., Shortliffe E. Production rules as a representation for a knowledge-based consultation program. Artif. Intell. 8, 1 (Feb. 1977).
4. Doyle J. Truth maintenance systems for problem solving. In: Proc. of the 5th International Joint Conference on Artificial Intelligence, 1977, p. 247.
5. Ernst G. W., Newell A. GPS: A Case Study in Generality and Problem Solving. Academic Press, Orlando, Fla, 1969.
6. Fikes R., Nilsson N. STRIPS: A new approach to the application of theorem proving to problem solving. Artif. Intell. 2, 3, 4 (Jan. 1971), 189—208.
7. Friedberg R. M. A learning machine. IBM J. Res. 2, 1 (Jan. 1958), 2—13.
8. Friedberg R. M., Dunham B., North J. H. A learning machine, p. II. IBM J. Res. 3, 3 (July, 1959), 282—287.
9. Green C. Theorem-proving by resolution as a basis for question answering systems. In: Machine Intelligence 4, B. Melter and D. Michie, Eds. University of Edinburgh Press, Edinburgh, 1969, p. 183—205.
10. Lifschitz V. Computing circumscription. In: Proc. of the 9th International Joint Conference on Artificial Intelligence, vol. 1, 1985, pp. 121—127.
11. McCarthy J. Programs with common sense. In: Proceedings of the Teddington Conference on the Mechanization of Thought Processes. Her Majesty's Stationery Office, London. Reprinted in: Semantic Information Processing, M. Minsky, Ed. M. I. T. Press, Cambridge, Mass., 1960.
12. McCarthy J., Hayes P. J. Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence 4, D. Michie, Ed. American Elsevier, New York, 1969.
13. McCarthy J. First order theories of individual concepts and propositions. In: Machine Intelligence 9, D. Michie, Ed. University of Edinburgh Press, Edinburgh, 1979.
14. McCarthy J. Some expert systems need common sense. In: Computer Culture: The Scientific, Intellectual and Social Impact of the Computer, vol. 426. Paggels, Ed. Annals of the New York Academy of Sciences, New York, 1983.

15. McCarthy J. Circumscription — A form of non-monotonic reasoning. *Artif. Intell.* 13, 1, 2 (Apr. 1980).
16. McCarthy J. Applications of circumscription to formalizing common sense knowledge. *Artif. Intell.* (Apr. 1986).
17. McDermott D., Doyle J. Non-monotonic logic I. *Artif. Intell.* 13, 1, 2 (1980), 41—72.
18. Newell A., Shaw J. C., Simon H. A. Preliminary description of general problem solving program-I (GPS-I). CIP Working Paper 7, Carnegie-Mellon Univ., Dec. 1957.
19. Newell A., Shaw J. C., Simon H. A. Report on a general problem-solving program for a computer. In: *Information Processing: Proceedings of the International Conference on Information Processing (Paris)*. UNESCO, 1960, pp. 256—264. (RAND P-1584, and reprinted in *Computers and Automation*, July 1959.)
20. Newell A., Shaw J. C., Simon H. A. A variety of intelligent learning in a General Problem Solver. In: *Self-Organizing Systems*, M. C. Yovits and S. Cameron, Eds. Pergammon Press, Elmsford, N. Y., 1960, pp. 153—189.
21. Newell A., Simon H. A. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, N. J., 1972.
22. Laird J. E., Newell A., Rosenbloom P. S. *Soar: An architecture for general intelligence*.
23. Reiter R. A logic for default reasoning. *Artif. Intell.* 13, 1, 2 (Apr. 1980).
24. Simon H. Still unsubstantiated rumor, 1960. GENERA [W86, JMC] TEXed on May 27, 1986, at 11:50 p.m.

1973

## Программист-навигатор

*Чарльз В. Бахман*

Ричард Каннинг, председатель комитета по присуждению Тьюринговской премии 1973 г., представляя эту лекцию 28 августа на ежегодной конференции Ассоциации вычислительных машин (АСМ) в Атланте, сказал следующее.

В последние пять — восемь лет в сфере вычислительной техники произошли важные изменения в области обработки данных. В самом начале развития информатики данные были тесно связаны с использующими их прикладными программами. Теперь ясно, что нужно разорвать эту связь. Мы хотим иметь данные, не зависящие от того, какая прикладная программа их использует, — т. е. данные, организованные и структурированные таким образом, чтобы они могли использоваться во многих приложениях и многими пользователями. То, к чему мы стремимся, — это *база данных*.

Работа по созданию баз данных сейчас находится в периоде младенчества. Но тем не менее уже известно от 1000 до 2000 систем управления базами данных, работающих по всему миру. Очень вероятно, что через десять лет число таких систем будет исчисляться десятками тысяч. Именно из-за большого количества инсталлированных систем влияние баз данных обещает быть огромным.

Человек, получающий премию Тьюринга в этом году, является настоящим пионером в области технологии создания баз данных. Ни один человек не повлиял так, как он, на развитие этой части информатики. Я выбрал три важнейших примера того, что он сделал.

Он был создателем и главным архитектором первой поступившей на рынок системы управления базами данных — The Integrated Data Store, разработанной в период с 1961 по 1964 г. [1, 2, 3, 4]. В настоящее время I-D-S является одной из наиболее широко используемых систем управления базами данных. Он был также одним из членов — учредителей группы по созданию базы данных CODASYL и работал в этой группе с 1964 по 1968 г. Технические требования, разработанные этой группой, выполняются и сейчас многими разработчиками

программ в различных частях света [5, 6]. Фактически сейчас эти требования являются единственной общепринятой основой архитектуры систем управления базами данных. Именно благодаря этому человеку эти требования после продолжительных дебатов и дискуссий воплотили многие из первоначальных идей I-D-S. И в-третьих, этот человек является создателем мощного метода представления отношений между данными — инструмента как для разработчиков баз данных, так и для разработчиков прикладных программ [7, 8].

Таким образом, его исследования продемонстрировали союз воображения и практики. Его глубокие работы уже оказали и будут оказывать значительное влияние на развитие информатики.

Я очень рад вручить премию Тьюринга 1973 г. Чарльзу В. Бахману.

Коперник полностью изменил наши взгляды на астрономические явления, предположив, что Земля вращается вокруг Солнца. Растет уверенность в том, что люди, занимающиеся обработкой информации, сильно выиграли бы, если бы восприняли радикально новую точку зрения, которая освобождает мышление прикладного программиста от централизма напоминающего устройства на магнитных сердечниках и дает ему свободу действовать как навигатору в базе данных. Для этого ему нужно, во-первых, освоить разнообразные навигационные умения и, во-вторых, выучить «правила движения», чтобы избежать конфликта с другими программистами, когда они будут одновременно находиться в информационном пространстве базы данных.

Новая точка зрения внесет такую же смуту в ряды программистов, какую в свое время гелиоцентрическая теория внесла в ряды старинных астрономов и теологов.

В этом году весь мир празднует пятистолетие со дня рождения Николая Коперника, знаменитого польского астронома и математика. В 1543 г. он опубликовал свою книгу «Об обращении небесных сфер», в которой изложил новую теорию относительного движения Земли, планет и Солнца. Она была в прямом противоречии с геоцентрической теорией Птолемея, появившейся на 1400 лет раньше.

Коперник создал гелиоцентрическую теорию, основная идея которой состоит в том, что планеты движутся вокруг Солнца по круговым орбитам. Эта теория в течение долгого времени подвергалась сильнейшей критике. Приблизительно через столет Галилею пришлось предстать перед судом инквизиции в Риме, и он был вынужден заявить, что больше не верит в теорию Коперника. Но даже это не успокоило инквизиторов, и Галилей был приговорен к бессрочному тюремному заключению, а книга Коперника была помещена в список запрещенных книг, в котором и находилась в течение последующих двухсот лет.

Я вспоминаю о Копернике сегодня, чтобы проиллюстриро-

вать параллель, которая, как я верю, существует в мире вычислительной техники, или, точнее, в мире информационных систем. Последние 50 лет мы работали с информационными системами птолемеевского типа. Эти системы и большинство размышлений о таких системах основывались на компьютероцентрических идеях. (Я не случайно говорю о 50, а не о 25 годах нашей истории, потому что я считаю, что современные информационные системы берут начало от первых счетно-аналитических машин (табуляторов), а не от первых вычислительных машин с запоминаемой программой.) Люди античности считали само собой разумеющимся, что Солнце вращается вокруг Земли; точно так же и люди античного периода развития информационных систем считали само собой разумеющимся, что через счетно-аналитическую машину должен проходить последовательный файл. Каждая из этих точек зрения являлась адекватной моделью действительности для того времени и места, когда и где была создана. Но через некоторое время каждая из них оказалась некорректной и неадекватной, и ее пришлось заменить на другую модель, которая более точно описывает окружающий мир и его поведение.

Коперник познакомил мир с новой точкой зрения и заложил фундамент современной небесной механики. Этот взгляд создал основу для понимания путей Солнца и планет по небу, ранее казавшихся таинственными. Новая основа для понимания существует и в области создания информационных систем. Она состоит в повороте от компьютероцентрических представлений к представлениям, в центре которых находится база данных. Новое понимание приведет к новому решению наших проблем, связанных с построением баз данных, и ускорит порождение *n*-мерных структур данных, являющихся наилучшей моделью сложностей реального мира.

Самые первые базы данных, реализованные с помощью технологии последовательного доступа к файлам, в которых носителями информации были еще перфокарты, не сильно изменились, когда их перенесли с перфокарт сначала на магнитную ленту, а затем на магнитный диск. Единственное, на что это хоть как-то повлияло, были размер файлов и скорость работы с ними.

Для технологии последовательного доступа к файлам техника поиска хорошо развита. Нужно начать со значения первичного ключа интересующей нас записи и считывать в основной памяти все записи подряд до тех пор, пока не встретится нужная запись или запись с большим значением первичного ключа. (Первичный ключ — это область в записи, которая делает эту запись уникальной внутри данного файла.) Первичными ключами являются номера ценных бумаг, налоговых кви-



танций, страховых свидетельств, банковских счетов. Почти все они без исключения являются искусственными атрибутами, специально предназначенными для удостоверения уникальности. Естественные атрибуты, как, например, имена людей, географические названия, даты, времена и количества, не являются достоверно уникальными и поэтому не могут быть использованы с этой целью.

Появление запоминающих устройств прямого доступа стало причиной переворота в мышлении, похожего на коперниковский. Направления, обозначаемые словами «в» и «из», поменялись на обратные. Если директива «ввод» в мире последовательного доступа к файлам означала «с ленты в вычислительную машину», то новая директива «ввод» означает «в базу данных». Эта революция в мышлении превращает программиста из неподвижного наблюдателя объектов, проходящих перед ним в памяти машины, в активного навигатора, который может перемещаться по базе данных вдоль и поперек по своему желанию.

Кроме того, с появлением запоминающих устройств с прямым доступом появились новые способы поиска записей по первичному ключу. Первый из них был назван рандомизацией или хэшингом. В нем происходит обработка первичного ключа с помощью специального алгоритма, который определяет область памяти, в которой с наибольшей вероятностью может находиться искомая запись. Если запись в этой области не найдена, применяется алгоритм переполнения (overflow algorithm) для нахождения других областей, в которых может храниться эта запись, если она вообще существует. Переполнение возникает при занесении записи в базу данных, если в этот момент область, где должна была бы находиться запись, оказывается заполненной.

Как альтернатива методу рандомизации был создан метод индексно-последовательного доступа. Для управления хранением и поиском записей в нем также применяются первичные ключи, но путем использования многоуровневых индексов.

Программист, перейдя от последовательной обработки файлов к хэшингу или к индексно-последовательному доступу, значительно уменьшает время доступа, ибо он теперь может добраться до интересующей его записи, не проходя последовательно всех предыдущих записей в файле. Однако он до сих пор находится в одномерном мире, так как имеет дело с одним первичным ключом, являющимся единственным инструментом для управления доступом.

С этого места я хочу начать обучение «полностью оперившегося» программиста-навигатора ориентированию в  $n$ -мерном пространстве данных. Но прежде, чем я смогу понятно опи-

сать этот процесс, я хочу пояснить, что же означают слова «управление базой данных».

Это понятие включает в себя все аспекты хранения, поиска, изменения и удаления данных в файлах, касающихся персонала или продукции, покупки авиабилетов или лабораторных экспериментов, — данных, которые используются неоднократно и обновляются всякий раз, как только становится доступной новая информация. Эти файлы с помощью некоторой запоминающей структуры и соответствующих драйверов преобразуются в информацию, носителями которой является магнитная лента или пакет дисков.

Управление базой данных включает в себя два основных вида деятельности. Первый — обработка запросов и поиск информации — обеспечивает доступ к сохраненным ранее данным, чтобы определить запомненное состояние некоей сущности или отношения в реальном мире. Эти данные могли быть записаны другой задачей несколько секунд, минут, часов или даже дней назад и были сохранены системой управления базой данных. Постоянной обязанностью системы управления базой данных является хранение информации с того момента, когда она была записана, до тех пор, когда она будет востребована запросом. Обработка запросов служит для того, чтобы находить информацию, необходимую для принятия решений.

Частью деятельности по обработке запросов является формирование сообщений. В давние годы запоминающих устройств с последовательным доступом и пакетной обработкой данных не было другой альтернативы, кроме создания и вывода на печать файлов в формате сообщений. Спонтанные требования: просмотреть банковский счет, инвентарную опись или план выпуска продукции не могли быть выполнены эффективно, потому что для отыскания каких-либо данных приходилось просматривать весь файл. Удельный вес такой формы обработки запросов уменьшается, и со временем она исчезнет совсем, кроме как для хранения архивов или для удовлетворения appetitов паркинсоновской бюрократии.

Вторым видом деятельности при управлении базой данных является обновление данных, что включает в себя ввод информации в базу данных, ее многократные изменения и, наконец, удаление ее из системы, когда она перестанет быть нужной.

Обновление данных является реакцией на те изменения в реальном мире, которые должны быть зарегистрированы. Прием на работу нового сотрудника приведет к появлению новой записи, которая должна быть сохранена. Уменьшение количества товаров на складах приведет к изменению инвентарной описи. Отмена предварительного заказа на билет на самолет

приведет к удалению записи. Все эти данные должны храниться и обновляться в ожидании следующих запросов.

Для сортировки файлов требовалось большое количество компьютерного времени. Сортировка файлов использовалась при сортировке транзакций, предшествующей пакетному последовательному обновлению базы данных, и при подготовке сообщений. Переход к обновлению базы данных в режиме оперативной обработки транзакций и к немедленному исполнению запросов на извлечение информации из базы данных и на подготовку отчетов уменьшает необходимость сортировки на уровне файлов.

А теперь давайте вернемся к нашему рассказу о программисте-навигаторе. Мы покинули его, когда он пытался воспользоваться методами рандомизации или индексно-последовательного доступа, основанными на использовании первичных ключей, чтобы ускорить процесс поиска и обновления информации в файле.

В дополнение к первичным ключам записей полезно иметь возможность находить запись по значениям некоторых других полей. Например, при планировании премий за 10 лет работы может понадобиться найти все такие записи о служащих, у которых значение поля «год приема на работу» было бы равно 1964. Такой метод доступа называется поиском по вторичному ключу. Количество записей, которые будут найдены по значению вторичного ключа, непредсказуемо и может изменяться от нуля до общего количества записей в файле. В противоположность этому по первичному ключу может быть найдена максимум одна запись.

С появлением поиска по вторичным ключам пространство данных, бывшее одномерным, приобретает дополнительные измерения, число которых равно числу полей в записи. Для файлов небольшого или среднего размера система управления базой данных может снабдить указателем каждое поле записи. Такие полностью индексированные файлы называются инвертированными файлами. Однако в больших активных файлах неэкономно индексировать каждое поле. Поэтому разумно выбрать те поля, содержимое которых будет наиболее часто использоваться, в качестве ключей (признаков, критериев) для поиска, и создать вторичные индексы только для этих полей.

Различия между файлом и базой данных не слишком ясны. Однако одно из них имеет непосредственное отношение к теме нашего разговора. В базах данных обычно бывает несколько или много различных типов записей. Например, в базе данных, содержащей сведения о персонале, могут быть записи с информацией о сотрудниках, различных подразделениях, квалификации, удержаниях из жалования, послужном списке и

образовании. Каждый тип записи имеет свой собственный уникальный первичный ключ, а все остальные ее поля являются потенциальными вторичными ключами.

В такой базе данных первичные и вторичные ключи находятся в интересных взаимоотношениях, когда первичный ключ записи одного типа играет роль вторичного ключа для записи другого типа. Возвращаясь к нашему примеру базы данных со сведениями о персонале, заметим, что поле с названием «код подразделения» встречается как в записях, относящихся к сотрудникам, так и в записях, относящихся к подразделениям. Это поле является одним из возможных вторичных ключей записей, относящихся к сотрудникам, и единственным первичным ключом записей о подразделениях.

Равноправность полей, содержащих первичные и вторичные ключи, отражает сложные взаимосвязи в реальном мире и дает возможность воссоздать эти взаимосвязи при компьютерной обработке информации. Использование одного и того же значения в качестве первичного ключа для одной записи и вторичного ключа для некоторого множества других записей является основной концепцией для возникновения и развития структурных наборов данных. Создатели системы Integrated Data Store (I-D-S) и других систем, построенных на основе тех же принципов, считают, что основным преимуществом этих систем для программиста является возможность объединять записи в структурные наборы данных, которые можно потом использовать как пути поиска. Все реализованные системы, созданные под руководством COBOL Database Task Group, являются системами этого класса.

Переход от системы файлов, каждый из которых содержит один тип записи, к базе данных с разными типами записей и структурированными наборами данных дает множество преимуществ. Одним из них является значительное увеличение эффективности, обусловленное использованием для поиска всех записей с определенным значением ключа структурированных множеств данных одновременно с первичными и вторичными ключами. При использовании структурированных наборов данных можно исключить всю избыточную информацию, что приводит к уменьшению необходимого объема памяти. Если избыточные данные были специально сохранены, чтобы увеличить скорость поиска, то их можно использовать, чтобы убедиться, что обновление какого-либо значения в одной записи влечет за собой изменения во всех соответствующих записях. Возможность так называемой «кластеризации» (объединения в кластеры), когда запись-хозяин и некоторые или большая часть записей данного множества физически хранятся в одном блоке или на одной странице, также увеличивает эффек-

тивность доступа. Системы такого типа работают с использованием виртуальной памяти с 1962 г.

Другое важное качественное преимущество базы данных — возможность выбирать порядок поиска записей внутри множества: по объявленному полю сортировки или же по времени ввода.

Чтобы подчеркнуть роль программиста как навигатора, давайте перечислим возможности поиска записи, которые имеются в его распоряжении. Это команды, с которыми он может обратиться к базе данных — один раз, многократно или комбинируя одну с другой, — когда он пробирается среди данных, чтобы ответить на запрос или обновить информацию.

1. Он может начать с начала базы данных или с любой известной ему записи и последовательно просматривать «следующие» записи до тех пор, пока не доберется до интересующей его записи или до конца.

2. Он может войти в базу данных с помощью ключа, обеспечивающего прямой доступ к физическому местоположению записи. (Ключ базы данных — это постоянный адрес в виртуальной памяти, приписанный записи при ее создании.)

3. Он может войти в базу данных по значению первичного ключа. (Как хэшинг, так и метод индексно-последовательного доступа дадут одинаковый результат.)

4. Он может войти в базу данных по значению вторичного ключа и последовательно просмотреть все записи, для которых в соответствующем поле стоит определенное значение вторичного ключа.

5. Он может начать с записи-хозяина некоторого множества и последовательно обратиться ко всем элементам этого множества. (Это эквивалентно превращению первичного ключа базы данных во вторичный.)

6. Он может начать с любого элемента множества и получить доступ как к следующему, так и к предыдущему элементу в этом множестве.

7. Он может начать с любого элемента множества и обратиться к записи-хозяину этого множества, преобразуя таким образом вторичный ключ базы данных в первичный.

Каждый из этих способов доступа интересен сам по себе, и все они очень полезны. Однако именно согласованное применение всех методов дает программисту широкие возможности войти в большую базу данных и перемещаться в ней, обращаясь только к тем записям, которые ему нужны при обработке запросов и обновлении информации для последующих запросов.

Чтобы проиллюстрировать, как обработка единственной транзакции может включать в себя определение пути внутри

базы данных, вообразив себе следующий сценарий. Трансакция содержит первичный ключ или просто ключ записи в базе данных, нужный для того, чтобы найти точку входа в базу данных. Эта запись будет использована, чтобы обратиться к другим записям (как к записи-хозяину, так и к элементам) множества. Каждая из этих записей в свою очередь может быть использована как точка выхода, чтобы получить доступ к другому множеству.

Например, рассмотрим требование составить список сотрудников какого-либо подразделения, если известен его код. Такой запрос может быть обращен к базе данных, содержащей только два типа записей: записи, содержащие сведения о сотрудниках, и записи, содержащие сведения о подразделениях. Для простоты будем рассматривать записи о подразделениях, в которых только два поля: код отдела, являющийся первичным ключом, и название отдела, которое является описанием. Рассмотрим также персональные записи, состоящие из трех полей, а именно: номер сотрудника по списку является первичным ключом записи, фамилия и имя сотрудника являются описанием, а код отдела, где работает сотрудник, является вторичным ключом, который управляет выбором множества и расположением записей в множестве. Одновременное использование кода отдела в обоих типах записей и объявление множества по этому ключу являются основой для создания и поддержания множественных отношений между записью, содержащей информацию о подразделении, и всеми персональными записями, содержащими сведения о сотрудниках этого подразделения. Таким образом, использование множества персональных записей обеспечивает механизм для того, чтобы после отыскания по первичному ключу соответствующей ему записи с информацией о конкретном подразделении немедленно выдать список сотрудников этого подразделения. Для этого не требуется обращаться ни к какой другой записи.

Добавление к записи о подразделении номера сотрудника, который руководит этим подразделением, в значительной мере расширяет навигационные возможности и служит основанием для создания множеств еще одного класса. При обращении к множеству из этого класса вызываются все записи о подразделениях, которыми руководит определенный сотрудник. Теперь единственное число — номер сотрудника или код отдела — дает точки входа в интегрированную структуру данных о предприятии. Если известны номер сотрудника и множество, состоящее из записей о подразделениях, которыми он руководит, то можно выдать список этих подразделений. Далее можно выдать список сотрудников каждого такого подразделения. Запрос о подразделениях, руководимых каждым из этих сотруд-

ников, можно повторять до тех пор, пока не будут перечислены все подчиненные сотрудники и подразделения. С другой стороны, с помощью той же самой структуры данных можно определить руководителя руководителя, руководителя руководителя руководителя и так далее — до президента компании.

Программиста, который научился действовать в  $n$ -мерном пространстве данных, ожидает еще немало опасностей и приключений. Как навигатор, он должен преодолевать плохо различимые мели и рифы в море информации, которые возникают потому, что он должен ориентироваться в общем пространстве базы данных.

Совместный доступ к базе данных — это новая и сложная разновидность мультипрограммного режима работы или режима работы с разделением времени, которые были созданы, чтобы обеспечить совместное, но независимое использование ресурсов вычислительной машины. При работе в мультипрограммном режиме программист, решающий свою задачу, не знает и не заботится о том, что его программа может быть не единственной в памяти компьютера, так как он уверен, что его адресное пространство не зависит от адресного пространства других программ. Обеспечить сохранность каждой программы и наилучшим образом использовать память, процессор и другие ресурсы — задача операционной системы. Совместный доступ — это специализированный вариант мультипрограммного режима работы, при котором совместно используемыми ресурсами являются записи в базе данных. Записи базы данных коренным образом отличаются от таких ресурсов, как оперативное запоминающее устройство или процессор, потому что их поля изменяют свои значения в процессе обновления данных и не возвращаются потом в исходное состояние. Поэтому задача, неоднократно использующая определенную запись базы данных, может обнаружить, что ее содержание или отношение принадлежности к множеству изменилось со времени предыдущего обращения. В результате алгоритм, осуществляющий сложные вычисления, может обращаться к довольно-таки нестабильной информации. Представьте себе, что вы решаете задачу с помощью сходящегося итерационного процесса, а переменные меняются случайным образом! Вообразите, что вы хотите подсчитать промежуточный баланс, а тем временем кто-то совершает разнообразные операции со счетами! Представьте себе две конкурирующие задачи в системе заказа авиабилетов, которые одновременно пытаются продать последний билет на рейс!

Первая реакция здравомыслящего человека — мысль, что распределенный доступ — это нонсенс и о нем надо забыть. Однако серьезные причины вынуждают его использовать и

развивать. Быстродействие процессоров, которые доступны сегодня и будут доступны в обозримом будущем, значительно превосходит быстродействие используемых запоминающих устройств прямого доступа. Более того, даже если быстродействие (эффективность) запоминающих устройств станет сравнимым с быстродействием процессоров, останется еще две важные причины стремления к разработке систем совместного доступа. Первая — это тенденция к объединению большого количества целевых файлов в несколько интегрированных баз данных; вторая — это тенденция к интегрированной обработке данных, при которой процессор может выполнять работу только в том темпе, в каком ему позволяют вручную вводимые сообщения. При отсутствии совместного доступа вся база данных будет заблокирована до тех пор, пока не закончится выполнение командного (пакетного, бат-) файла или транзакции и их взаимодействие с пользователем.

Реализация современных запоминающих устройств сильно зависит от принятой схемы их использования. Если она представляет собой чередующуюся последовательность вида: обращение к памяти, обработка, обращение к памяти, обработка и т. д. — и выполнение каждого обращения зависит от интерпретации предыдущего, то эффективность будет очень мала. В мультипрограммном режиме генерируется много независимых обращений к памяти, и они могут быть исполнены параллельно, потому что направлены разным запоминающим устройствам. Более того, когда существует очередь обращений к одному и тому же запоминающему устройству, его производительность в настоящее время может быть значительно увеличена с помощью методов предварительной обработки данных, уменьшающих время поиска и ожидания. Возможность увеличения пропускной способности является очень весомым доводом в пользу развития систем совместного доступа.

Из двух основных функций системы управления базой данных — обработка запросов и обновления данных — только вторая может быть причиной затруднений при работе в режиме совместного доступа. Неограниченное количество задач может одновременно извлекать информацию из базы данных. Но как только одна из задач начинает изменять данные, возникает возможность ошибок. В результате обработки транзакции может потребоваться изменение всего нескольких записей из тысяч или миллионов, находящихся в базе данных. Поэтому сотни задач могли бы выполнять транзакции одновременно без возникновения конфликтных ситуаций. Однако наступит момент, когда две задачи одновременно захотят работать с одной записью.

Двумя основными причинами трудностей при реализации



совместного доступа являются интерференция и порча данных. Интерференция — это отрицательное влияние изменения информации, производимого одной задачей, на результат работы другой. Я уже приводил пример, иллюстрирующий проблему интерференции, когда одна задача подсчитывает промежуточный баланс, в то время как другая выдает транзакции, приводящие к изменению счетов. Если в процессе решения задачи на нее влияет другая, то для получения правильного результата первая задача должна быть прервана и запущена снова. Все выходные данные, полученные при предыдущем проходе, нужно удалить, так как будут выданы новые. *Порча данных* — это отрицательный эффект, который происходит в результате комбинации двух событий, а именно когда вторая задача прервана, а ее выходные данные (т. е. изменения в базе данных или какие-либо сообщения) уже считаны нашей первой задачей. Прерванная задача и ее выходная информация должны быть удалены из системы. Более того, задачи, «испорченные» этой информацией (на работу которых повлияла эта информация), должны быть также прерваны и запущены повторно, чтобы данные, к которым они обращаются, были корректны.

Критическим вопросом при решении проблем совместного доступа является «глубина видимости», которую следует предоставить прикладному программисту. Версия I-D-S с совместным доступом, разработанная Weyerhaeuser Company, основывается на предпосылке, что программист вообще не должен знать о проблемах совместного доступа. В этой системе автоматически блокируются каждая измененная запись и каждое сообщение, сформированное задачей, до тех пор, пока не произойдет нормальное окончание этой задачи. Таким образом проблема порчи данных полностью снимается. Побочным эффектом такого динамического блокирования информации является возможность возникновения взаимоблокировки (зависания), когда две или больше задач ждут окончания работы друг друга, чтобы получить доступ к желаемой записи. Система I-D-S реагирует на возникновение взаимной блокировки следующим образом: она прерывает задачу, послужившую причиной этой ситуации, восстанавливает записи, измененные этой задачей, и делает эти записи доступными для всех задач, находящихся в состоянии ожидания. Затем перезапускается прерванная задача.

Возникают ли в действительности ситуации взаимоблокировки? По последним сведениям в системе Weyerhaeuser, ориентированной на обработку транзакций, около 10% всех запущенных задач бывают прерваны из-за взаимоблокировки. За час выбрасываются и перезапускаются приблизительно 100 задач. Ужасно ли это? Слишком ли неэффективно? На эти во-

просы трудно ответить, так как наши представления о стандартах эффективности работы таких систем четко не определены. Более того, результаты зависят от применений. Эффективность системы Weyerhaeuser I-D-S составляет 90% в терминах успешно завершенных заданий. Однако действительно важными вопросами являются следующие:

— К увеличению или к уменьшению количества задач, успешно завершенных в течение часа, привел бы отказ от совместного доступа?

— Не была ли бы более эффективной другая стратегия, основанная не на том, чтобы избегать порчи данных, а на том, чтобы своевременно определять такие ситуации?

— Не повысится ли эффективность, если программист будет знать о проблемах совместного доступа и учитывать это обстоятельство в своих программах?

Все эти вопросы становятся насущными для программиста-навигатора и для людей, которые создают и развивают средства навигации.

Я считаю, что сейчас для прикладного программиста настало время отказаться от представлений, в которых центром всего является память компьютера, принять вызов и воспользоваться возможностями навигации в  $n$ -мерном пространстве информации. Системы математического обеспечения для поддержки таких возможностей существуют сегодня и становятся все более доступными.

Бертран Рассел, знаменитый английский математик и философ, однажды сказал, что теория относительности потребовала изменений в нашей воображаемой картине мира. Похожее изменения должны произойти и в нашей воображаемой картине мира информационных систем.

Основная проблема — это переориентация мышления людей, занимающихся проблемами обработки информации. Это не только программисты, но создатели прикладных систем, ставящие задачи для прикладного программирования, и системные программисты, которые будут придумывать и реализовывать операционные системы, системы обмена сообщениями и системы управления базами данных завтрашнего дня.

Коперник заложил основы небесной механики более 400 лет назад. Это та самая наука, которая дает минимальные с точки зрения энергии решения при планировании траектории полетов к Луне и другим планетам. Нужно развивать такой же научный подход, результатом которого были бы энергетически минимальные решения задачи доступа в базах данных. Эта проблема вдвойне интересна, так как она включает в себя как задачи перемещения по уже существующим базам данных, так и вопрос, как построить базу данных и затем ее изменять, что-

бы она наилучшим образом соответствовала изменяющимся моделям доступа. Вы можете себе представить реконструкцию Солнечной системы с целью минимизировать время полета между планетами?

Необходимо, чтобы изучение механизмов структурирования информации развивалось как инженерная дисциплина, основанная на главных принципах проектирования. Необходимо, чтобы оно могло стать учебной дисциплиной и ее преподавали. Стоимость оборудования для баз данных, которые будут инсталлированы к 1980 г., оценивается в 100 млрд. долл. (на основе данных за 1970 г.). Далее, ожидается, что отсутствие эффективной стандартизации добавит еще 20% или 20 млрд. долл. к этой сумме. Следовательно, было бы разумнее отказаться от консерватизма, эмоций и теологических аргументов, замедляющих прогресс в настоящее время. В университетах проблемы структурирования информации в значительной степени игнорировались и предпочтение отдавалось задачам, которые лучше подходят для курсовых и дипломных работ. Создание большой базы данных — это дорогостоящая программа, и университетский бюджет просто не может себе позволить ее финансирование. Поэтому для финансирования и обеспечения необходимых для достижения прогресса мощностей нужно создавать совместные программы для университетов и промышленности или для университетов и правительственных учреждений. В системе Weyerhaeuser содержится достаточно материала для полудюжины диссертаций, который ждет, чтобы кто-нибудь пришел и откопал его. Я не имею в виду исследования по созданию новых алгоритмов рандомизации. Я имею в виду исследование по структурированию около миллиарда знаков информации о реальном бизнесе, организованных в самую безупречную из структур данных, известных на сегодняшний день.

Проблемой является и публикация технической литературы. Проще всего опубликовать свою работу в периодических выпусках тематических групп SIGBDP и SIGFIDET Ассоциации вычислительных машин (ACM). Правила реферирования статей и обычная практика публикации в журнале Communications of the ACM приводят к тому, что от получения статьи до ее издания проходит от одного года до полутора лет. Прибавьте к этому время, необходимое автору для подготовки своих идей к печати, и вы получите задержку между получением важных результатов и первой их публикацией по меньшей мере в два года.

Может быть, самым большим препятствием для прогресса является отсутствие у большей части пользователей доступной информации о базах данных, которое является следствием то-

го, что интересы рынка доминируют над интересами отдельного пользователя. Если бы программисты предоставили свой опыт, требования и средства решения задач для действительно открытого обмена информацией, изменения происходили бы гораздо быстрее. Последняя акция SHARE — открыть членство для всех организаций, продающих программы, и всех пользователей — является важным шагом вперед. Рабочая конференция по системам управления базами данных, спонсором которой был SHARE, стала ареной свободной дискуссии, в которой пользователи всех видов оборудования и всевозможных баз данных могли рассказать о своем опыте и своих нуждах.

Все расширяющийся диалог начался. Я очень хочу, чтобы он продолжался, и надеюсь, что так и будет. Я уверен, что если продолжать в том же духе, если никто не будет стремиться доминировать в области идей, то мы сможем обеспечить программиста эффективными средствами навигации.

#### ЛИТЕРАТУРА

1. A general purpose programming system for random access memories (with S. B. Williams). Proc. AFIPS 1964 FJCC, Vol. 26, AFIPS Press, Montvale, N. J., pp. 411—412.
2. Integrated Data Store. DPMA Quarterly (Jan. 1965).
3. Software for random access processing. Datamation (Apr. 1965), 36—41.
4. Integrated Data Store—Case Study. Proc. Sec. Symp. on Computer-Centered Data Base Systems sponsored by ARPA, SDC, and ESD, 1966.
5. Implementation techniques for data structure sets. Proc. of SHARE Working Conf. on Data Base Systems, Montreal, Canada, July 1973.
6. The evolution of Data Structures. Proc. NordDATA Conf., Aug. 1973, Copenhagen, Denmark, pp. 1075—1093.
7. Data structure diagrams. Data Base 1, 2 (1969), Quarterly Newsletter of ACM SIGBDP, pp. 4—10.
8. Set concepts for data structures. In Encyclopedia of Computer Science, Amerback Corp. (to be published in 1974).

#### ПОСТСКРИПТУМ

ПРОГРАММИСТ КАК НАВИГАТОР, АРХИТЕКТОР, СВЯЗИСТ,  
СОЗДАТЕЛЬ МОДЕЛЕЙ, СОТРУДНИК ЭКСПЕРТНОЙ СИСТЕМЫ  
И РУКОВОДИТЕЛЬ

Чарльз В. Бахман  
Bachman Information Systems, Inc.

С момента написания Тьюринговской речи, которая называлась «Программист-навигатор», прошло 13 лет. Использование баз данных стало обычным, даже популярным занятием. Некоторые программисты используют средства навигации. Остальные пытаются это делать. Я приложил значительные усилия, чтобы доказать преимущества сетевой модели данных и ее расширения с целью увеличить ее выразительные возможности [1, 2, 3, 4]. Горячие и яростные споры и дискуссии, касающиеся моделей данных, к настоящему времени в значительной степени остались позади. Сейчас единст-

венное, с чем все согласны — тот факт, что можно с пользой работать с базами данных, основанными на любой из общепринятых моделей данных, и даже с теми, которые не имеют явного отношения ни к какой конкретной модели данных.

## ПРОГРАММИСТ КАК АРХИТЕКТОР

Изучение архитектуры компьютерных информационных систем далеко продвинулось за это время. Два проекта, каждый из которых по-своему важен, способствовали привлечению внимания к этому вопросу. Группа ANSI/X3/SPARC по изучению систем управления базами данных (1972—1977 гг.) опубликовала свои исследования [5] по архитектуре систем хранения и поиска данных. Это была одна из первых попыток четко понять и описать различные уровни работы человека и программного обеспечения в процессе хранения и поиска информации. Более того, были выделены и описаны интерфейсы между различными модулями математического обеспечения и людьми, которые с ними работают (администраторами, создателями баз данных и программистами). Очень важно, что в этой работе описаны как административный интерфейс, так и интерфейс времени исполнения. Этот проект способствовал развитию понятия концептуальной схемы как описания информационных структур, сделанного на более высоком уровне абстракции и не зависящего от способа представления информации.

## ПРОГРАММИСТ КАК СВЯЗИСТ

Подкомитет ISO/TC97/SC16 Международной организации по стандартизации (ISO) создал (1979—1982 гг.) Эталонную модель взаимосвязи открытых систем. Эталонная модель — это эталонная схема архитектуры систем обмена информацией, разработанная в виде международного стандарта [7], чтобы она служила контролирующей и объединяющей моделью для последующей серии более подробных стандартов. В этой архитектуре определяется семь иерархических уровней обработки данных, поддерживающих обмен между прикладными процессами. Каждый уровень описывается в терминах «административных логических объектов»<sup>1</sup>, «обрабатывающих логических объектов», «услуг» и «протоколов».

Для обрабатывающих логических объектов каждого уровня должны быть выделены и стандартизованы следующие четыре вида интерфейсов:

- (1) услуги, которые обрабатывающий объект предоставляет обрабатывающим объектам, лежащим на один уровень выше;
- (2) протокол обмена, с помощью которого обрабатывающий объект общается с другими обрабатывающими объектами одного с ним уровня;
- (3) использование обрабатывающими объектами данного уровня услуг, предоставляемых обрабатывающими объектами, лежащими на один уровень ниже;
- (4) административный протокол, посредством которого данный обрабатывающий объект контролируется административными объектами своего уровня.

<sup>1</sup> Сочетание «логический объект» употребляется в стандарте ISO/TC97 для обозначения активного элемента, который играет некую роль в процессе обмена. Я использовал прилагательные «обрабатывающий» и «административный» для различия объектов, действующих во время обмена и при инициализации (set-up-time entities). Такое использование слова «объект» отличается от принятого при моделировании данных, где «объект» означает что-то, что существует и о чем что-либо известно.

Подробные стандарты, разработанные последовательно для каждого уровня, касаются индивидуальных протоколов, услуг и использования услуг.

Глубину предвидения и масштаб этой работы можно в какой-то мере оценить, вспомнив некоторые из дискуссий об адресуемости (addressability). Как велико должно быть адресное пространство, чтобы определить все обрабатывающие логические объекты, которые могут захотеть обмениваться информацией? В результате одной из дискуссий возник следующий сценарий:

К концу 2000 года численность населения Земли составит около 10 миллиардов человек (10 миллиардов адресов).

Предположим, что каждого из этих людей будет обслуживать в среднем 100 роботов (1 триллион адресов).

Для учета непредвиденных случайностей, а также продолжительности срока полезного использования адресного пространства в 25 лет, умножим еще на 10 (10 триллионов адресов).

Будем предполагать, что присвоение адресов происходило в ходе политических процессов начиная с образования Организации Объединенных Наций, и что 99 процентов адресов на самом деле недоступно для обмена на уровне прикладных программ (1 квадрильон адресов).

Следовательно, один квадрильон адресов — это правильный порядок величины адресного пространства для рассмотренного случая. В десятичной системе такое число представляется единицей с 15 нулями, в двоичной это единица и приблизительно 50 нулей.

В этом году работа ISO по созданию стандартов для взаимосвязи открытых систем получила дополнительную поддержку в результате создания в Соединенных Штатах Корпорации по открытым системам (Corporation for Open Systems). COS — это организация, деятельность которой охватывает всю индустрию создания и использования баз данных: пользователей, связистов и производителей. Она образована для поддержки выполнения стандартов ISO и для того, чтобы новое или модифицированное оборудование могло быть проверено на соответствие стандартам ISO.

Автор, делая доклад для технического комитета ISO/TC97 в качестве председателя подкомитета ISO/TC97/SC16, рекомендовал комитету TC97 разрабатывать «эталонную модель для автоматизированных информационных систем» [8, 9]. Эту расширенную модель можно было бы использовать, чтобы включить всю работу комитета ISO/TC97 по автоматическим и информационным системам в перспективу дальнейшего развития и таким образом выделить наиболее критические с точки зрения стандартизации её участки.

В 1984—1985 гг. структура подкомитетов ISO/TC97 изменилась, и был создан новый подкомитет ISO/TC97/SC21, который взял на себя не только все функции SC16, но и дополнительные обязанности, касающиеся архитектуры систем хранения и поиска информации. Со временем в его компетенцию будут входить также вопросы сохранности данных и их защищенности от несанкционированного доступа, потому что невозможно создать законченную схему архитектуры систем хранения, поиска и обмена информацией в отрыве от проблем сохранности данных и их защиты (секретности).

## ПРОГРАММИСТ КАК СОЗДАТЕЛЬ МОДЕЛЕЙ

За эти 13 лет я потратил довольно много времени на расширение работ над концептуальной схемой в группе по изучению СУБД (ANSI/SPARC Study Group on DMBS), сочетая эту деятельность с работой в области обмена информацией и методов формальных описаний. Вначале работа над

концептуальной схемой ограничивалась только информацией и форматами данных, касающимися бизнеса, как теми, которые хранятся в файлах и базах данных (внутренняя схема), так и представленными в программах (внешняя схема). Моей целью было расширить эту абстракцию и включить в нее описания всех активных агентов (людей, компьютерных программ и физических процессов), использующих информацию, а также каналов связи, которыми они пользуются, и сообщений, которыми они обмениваются.

Я стремился еще больше расширить эту абстракцию и включить в нее правила, которым подчиняется поведение объектов (агентов), использующих информацию. Такие расширенные концептуальные схемы получили название **моделей предприятия (enterprise models)** или **моделей бизнеса (business models)**.

Зачем строить модель бизнеса? Во-первых, она является таким способом описания требований к организации процесса обработки информации, который одинаково понятен и пользователям, и людям, занимающимся обработкой информации. Во-вторых, она нужна как основа для автоматизации процесса создания прикладного программного обеспечения. Прикладным программным обеспечением я называю совокупность, включающую в себя описания базы данных и файлов, прикладные программы, а также контрольные параметры оборудования, необходимые для инсталляции необходимых файлов и программ и управления их работой.

Операция перевода модели бизнеса в множество прикладных программ, реализующих и поддерживающих эту модель — это операция, переводящая вопрос «что?» в мире бизнеса в вопрос «как?» в мире компьютеров и обмена информацией. Для такого перевода требуется три дополнительных элемента, лежащих за пределами формального определения модели бизнеса:

1. Нужна информация о количествах, скоростях и временах отклика как параметрах, которым должна удовлетворять модель.

2. Нужна информация об имеющихся в наличии процессорах, запоминающих устройствах, средствах связи и информация об имеющихся компиляторах, СУБД, системах передачи информации, мониторах, обрабатывающих транзакции, и операционных системах.

3. Необходимы также экспертные знания для того, чтобы оценить рабочие параметры и эксплуатационные характеристики имеющегося в наличии оборудования и программного обеспечения, и затем определить способ его наиболее эффективного использования с учетом функциональных и количественных ограничений.

Таковыми экспертными знаниями по исполнению и оптимизации обладают конкретные люди — создатели баз данных, прикладные программисты, системные программисты. Лучшие из них превосходно справляются со своей работой, но результаты работы многих других разочаровывают. Все это дорого и занимает гораздо больше времени, чем хотелось бы.

## ПРОГРАММИСТ КАК СОТРУДНИК ЭКСПЕРТНОЙ СИСТЕМЫ

Недостаток специалистов достаточно высокой квалификации побудил нас начать исследования по автоматизации работы создателей баз данных, прикладных и системных программистов. Такая автоматизация очень сложна, так как процесс перевода модели бизнеса в эффективно работающие прикладные программы не является вполне детерминированным. Часто существует несколько альтернативных подходов, каждый со своей динамикой развития и затратами. В этом случае необходимо проводить экспертизу и принимать решение. Эта трудность привела к тому, что при оценке инструментальных средств и оборудования используются методы, разработанные в мире искусственного интеллекта, в котором большое внимание уделяется областям, связанным с применением неполных знаний (*imperfect knowledge*).

В мире искусственного интеллекта с его программными комплексами,

основанными на использовании знаний, существует значительный опыт разработки интерактивных систем, в которых постоянно работающий человек-эксперт может сотрудничать с «клонированным» экспертом, содержащимся в комплексе программ, и решать задачи, которые трудно решить другим способом. Вместе они могут сделать все необходимое для перехода с абстрактного концептуального уровня на физический, учитывая все проблемы и возможности конкретной реализации.

## ПРОГРАММИСТ КАК РУКОВОДИТЕЛЬ

Есть причины думать, что «клонированные» эксперты, содержащиеся в системах, использующих знания (экспертных системах), будут совершенствоваться со временем. В процессе этого совершенствования роль резидентного эксперта-человека будет изменяться, и из соавтора экспертной системы он будет превращаться в ее руководителя. Как руководитель он будет контролировать работу экспертной системы и проверять, все ли рабочие режимы и условия эксплуатации предусмотрены. После проверки и просмотра всех возможных модификаций эксперт как руководитель должен будет подписать окончательный проект, так же как руководитель группы инженеров подписывает результат работы своих подчиненных. При использовании информационных систем в области бизнеса ничего не может быть сделано без того, чтобы кто-нибудь это не проверил и не нес за это ответственность.

## ЗАКЛЮЧЕНИЕ

Есть что-то поэтическое в том, что методы, использующиеся при создании баз данных, соединяются с методами из области искусственного интеллекта. Поэтическое, потому что уже первые упоминания (1960 г.) об обработке списков в литературе по системам искусственного интеллекта послужили основой для создания связанных списков, которые были первым и до сих пор остаются самым часто используемым средством реализации баз данных. Путаница с понятием структурного набора данных и наиболее распространенным способом его воплощения внушает беспокойство. Существует много хорошо известных способов реализации структурных наборов данных [10], каждый со своими рабочими характеристиками, и все они обеспечивают адекватную поддержку функциональных качеств множества.

Интересно, удастся ли методам экспертной оценки реализаций из мира баз данных существенно повлиять на мир LISP'a и искусственного интеллекта, как это происходит в коммерческих применениях, где базы знаний велики и распределены по многочисленным взаимодействующим рабочим станциям с системами искусственного интеллекта. В этом случае эффективность и время отклика зависят от успешной работы совместно используемой системы с базами знаний распределенной виртуальной памяти.

## ЛИТЕРАТУРА

1. Bachman, C. W. Why restrict the modeling capability of the CODASYL data structure sets? in Proceedings of the AFIPS National Computer Conference, vol. 46. AFIPS Press, Reston, Va., 1977.
2. Bachman, C. W., and Daya, M. The role concept in data models. In Proceedings of the 3rd Very Large Database Conference, 1977.
3. Bachman, C. W. The structuring capabilities of the molecular data model (partnership data model). In Entity-Relationship Approach to Software Engineering. Elsevier Science, New York, 1983.
4. Bachman, C. W. The partnership data model. Presented at the Fall 1983 IEEE Computer Conference (Washington, D. C.).



5. ANSI/X3/SPARC/Study Group — Database Management Systems. Framework Report on Database Management Systems. AFIPS Press, Reston, Va., 1978.
6. ISO/TC97/SC5/WG3. Concepts and terminology for the conceptual schema. January 15, 1981.
7. ISO. Computers and Information Systems — Open Systems Interconnection Reference Model. Standard 7498. American National Standards Institute, New York, N. Y.
8. Bachman, C. W. The context of open systems interconnection within computer-based information systems. In *Proceedings of Gesellschaft fur Informatik*, Jan. 1980.
9. Bachman, C. W., and Ross, R. G. Toward a more complete reference model of computer-based information systems. *J. Comput. Standards* 1 (1982); also published in *Comput. Networks* 6 (1982).
10. Bachman, C. W. Implementation of techniques for data structure sets. In *Proceedings of SHARE Workshop on DataBase Systems* (Montreal, Canada, July, 1973).

1975

# Информатика

## как эмпирическое исследование:

### СИМВОЛЫ И ПОИСК

*Аллен Ньюэлл, Херберт Саймон*

Тьюринговская премия 1975 г. была присуждена Аллену Ньюэллу и Херберту Саймону на ежегодной конференции АСМ, состоявшейся 20 октября в Миннеаполисе. Представляя лауреатов, Бернард А. Галлер, председатель Комитета по присуждению премий им. Тьюринга, заявил:

Я рад, что могу сегодня наградить Тьюринговской премией двух моих давних друзей, профессоров Аллена Ньюэлла и Херберта Саймона, из Университета Карнеги — Меллона.

В результате совместной научной работы, продолжающейся более двадцати лет, начатой вместе с Дж. Шоу из Рэнд Корпорейшн и продолженной в сотрудничестве со многими студентами и сотрудниками Университета Карнеги — Меллона, они внесли основополагающий вклад в исследования по искусственному интеллекту, психологию познания и в обработку списков.

В области искусственного интеллекта они способствовали становлению этой дисциплины как арены интенсивной научной работы, развитию эвристического программирования вообще и эвристического поиска, анализа средств и целей, а также методов индукции в особенности, продемонстрировав достаточность этих механизмов для решения интересных задач.

В психологии они были главными пропагандистами идеи о том, что человеческое сознание может быть описано как символическая система, и они разработали подробные теории решения задач человеком, вербального обучения и индуктивного поведения в ряде предметных областей, используя компьютерные программы для воплощения этих теорий и моделирования поведения человека.

Они несомненно являются изобретателями обработки списков, внесшими важный вклад как в технологию разработки программного обеспечения, так и в создание концепции компьютера как системы для манипулирования символическими структурами, а не просто процессора для обработки числовых данных.

Это награда почетна для профессоров Ньюэлла и Саймона, но и для нашей Ассоциации большая часть, что мы можем включить их имена в список наших лауреатов, поскольку это повышает престиж и значимость Тьюринговской премии АСМ.

\* \* \*

Информатика — это изучение явлений, связанных с вычислительными машинами. Основатели нашего общества отлично понимали это, назвав его Ассоциацией вычислительной техники (Association for Computing Machinery). Машина (причем не только аппаратура, но живая, запрограммированная машина) — вот тот организм, который мы изучаем.

Данная Тьюринговская лекция — десятая по счету. Девять человек, которые были нашими предшественниками на этой трибуне, изложили девять различных точек зрения, поскольку на изучаемый нами организм, вычислительную машину, можно смотреть с разных сторон, и исследовать его на разных уровнях. Мы глубоко признательны за возможность присутствовать здесь сегодня и изложить еще одну точку зрения, ту, которая послужила основой научной работы, за которую мы удостоены этой награды. Мы будем говорить об информатике как об эмпирическом исследовании.

Наш подход — лишь один из многих возможных; предыдущие лекции вполне прояснили это. Однако все лекторы-лауреаты, вместе взятые, все же не могут очертить весь круг задач нашей науки. Многие фундаментальные ее аспекты остались за пределами этих десяти лекций. И если когда-нибудь и наступит время — а это наверняка будет нескоро — когда на карте принадлежащих информатике земель не останется белых пятен и она будет обсуждена со всех возможных сторон, — тогда-то и придет пора начать все сначала. Ведь такому зайцу, как лектор, приходится каждый год предпринимать спринтерский бросок, чтобы догнать черепаху научного и технического прогресса и преодолеть то расстояние, которое она успела за год проползти своим неторопливым, размеренным шагом по стезе малых, постепенных улучшений. Каждый год порождает новое отставание и требует нового спринтерского броска, так как в науке последнего слова не бывает.

Информатика — это эмпирическая дисциплина. Ее можно было бы назвать экспериментальной наукой, если бы, как и в случае с астрономией, экономикой и геологией, некоторые из ее уникальных форм наблюдения и практики не исключали их подгонку под узкий стереотип экспериментального метода. Тем не менее, это все же эксперименты. Каждая новая построенная машина — это эксперимент. В самом деле, конструирование машины задает природе некий вопрос; мы слушаем ответ, наблюдая машину в работе, и анализируем его, применяя все имеющиеся в нашем распоряжении средства измерения и анализа. Ни машины, ни программы не являются черными ящиками; они создаются и проектируются людьми, мы можем вскрыть их и посмотреть, что там внутри. Мы можем сопоставить их устройство с их поведением и многому научиться

на одном-единственном эксперименте. Нам не нужно создавать 100 экземпляров, скажем, доказывателя теорем, чтобы статистически продемонстрировать, что он не может избежать комбинаторного взрыва при поиске решения тем способом, на который мы рассчитывали. Изучение программы после нескольких прогонов позволяет отыскать ошибку и перейти к новой попытке.

Мы делаем компьютеры и пишем программы по многим причинам. Мы строим их, чтобы они служили обществу, решали его экономические задачи. Но как представители фундаментальной науки мы строим компьютеры и пишем программы для того, чтобы открывать новые явления и исследовать те явления, которые нам уже известны. Общество часто оказывается неспособным это понять; оно считает, что компьютеры и программы следует разрабатывать лишь ради экономической выгоды, которую они могут принести (или же в качестве промежуточных этапов процесса разработки, ведущего к такому использованию). Необходимо понимать, что явления, связанные с компьютерами, глубоки и загадочны, и прояснение их природы требует большой экспериментальной работы. Нужно осознать, что, как и во всякой науке, преимущества, приобретаемые в результате экспериментирования и понимания, реализуются в постоянном обретении новых методов, и именно эти методы позволяют создавать практически полезные инструменты, помогающие обществу достигать своих целей.

Однако смысл нашего выступления не в том, чтобы искать понимания со стороны общественности, далекой от наших проблем. Мы хотим проанализировать один аспект нашей науки, а именно развитие нового фундаментального знания путем эмпирического исследования. Лучше всего сделать это на конкретных примерах. Нам простят, если, учитывая повод, по которому мы приглашены здесь выступать, мы выберем примеры из области наших собственных исследований. Как станет ясно, эти примеры охватывают всю историю разработки искусственного интеллекта, особенно ее ранние этапы. Они основаны на гораздо более обширном материале, чем наши личные достижения. Даже там, где речь будет идти о наших собственных разработках, нужно помнить, что мы работали не одни. Из наших сотрудников следует особенно отметить Клиффа Шоу, вместе с которым мы образовали группу из трех человек и проработали совместно в течение всего волнующего периода конца 50-х годов. Но мы также работали вместе с очень многими студентами и сотрудниками Университета Карнеги — Меллона.

Недостаток времени позволяет выбрать лишь два примера. Первый из них — разработка понятия символьной системы.

Второй — разработка понятия эвристического поиска. Оба этих понятия очень важны для углубления нашего понимания того, как обрабатывается информация и как достигается интеллектуальность. Однако они не могут даже приблизительно охватить все направления работ по искусственному интеллекту, хотя они представляются нам полезными для демонстрации характера фундаментального знания в этой области информатики.

## 1. СИМВОЛЫ И ФИЗИЧЕСКИЕ СИМВОЛЬНЫЕ СИСТЕМЫ

Одним из важнейших вкладов в понимание информатики стало объяснение, причем на весьма фундаментальном уровне, что представляют собой символы. Это объяснение является научным утверждением о том, что такое Природа. Оно выведено опытным путем в ходе длительного и постепенного развития.

Символы лежат в основе разумных действий, которые, конечно, являются главным предметом изучения искусственного интеллекта. По этой причине это центральный вопрос всей информатики. Любая информация обрабатывается компьютерами для достижения некоторых целей, и мы оцениваем интеллектуальность системы по ее способности достигать заявленных целей в условиях изменчивости, трудностей и сложностей, порождаемых характером поставленных задач. Этот общий вклад информатики в достижение интеллектуальности затемняется, если поставленные задачи ограничены чрезмерно упрощенной постановкой, потому что тогда все разнообразие и изменчивость данных могут быть точно предвидены заранее. Это становится более очевидным, когда мы используем компьютеры при решении более широких, сложных и требующих обширных знаний проблем, когда мы пытаемся сделать их своими слугами, способными самостоятельно справляться со всеми неожиданностями реального мира.

Понимание того, каким требованиям должна удовлетворять система, чтобы ее действия были разумными, возникает не сразу. Оно многосоставно, поскольку никакое элементарное свойство не может быть причиной интеллектуальности во всех ее проявлениях. Не существует «принципа интеллектуальности», также как не существует «принципа жизненности», несущего в себе по самой своей природе сущность жизни. Но отсутствие какого-то простого «бога из машины» не означает, что не существует никаких структурных особенностей, необходимых для интеллектуального поведения. Одно из таких необходимых свойств — способность хранить символы и манипулировать

ими. Чтобы поставить этот вопрос научно, можно перефразировать название известной статьи Уоррена Маккалоха [3]: что есть символ, что интеллект может использовать его, и что есть интеллект, что он может использовать символы?

### КАЧЕСТВЕННЫЕ СТРУКТУРНЫЕ ПРИНЦИПЫ

Любая наука описывает сущность систем, которые она изучает. Эти описания по своей природе неизменно оказываются **качественными**, поскольку они устанавливают рамки, в которых можно развивать более детализированное знание. Сущность этих описаний зачастую можно выразить в очень кратких и очень общих утверждениях. Из-за ограниченной конкретности этих утверждений их можно было бы считать мало что прибавляющими к общей сумме научных знаний, если бы история не свидетельствовала о величайшей важности таких результатов.

**Клеточная теория в биологии.** Хороший пример качественного структурного принципа — клеточная теория, утверждающая, что основным структурным элементом всех живых организмов является клетка. Клетки бывают очень разными, но все они содержат ядро, окруженное цитоплазмой, а снаружи все это заключено в клеточную мембрану. Но эта внутренняя структура не является частью описания, которое мы называем клеточной теорией в ее первоначальном виде; это последующее уточнение, появившееся в результате интенсивных исследований. Клеточная теория почти целиком содержится в приведенном выше утверждении и некоторых достаточно приблизительных представлениях о том, какого размера могут быть клетки. Однако воздействие этого принципа на биологию было огромным, и замешательство среди биологов, предшествующее его постепенному признанию, было велико.

**Тектоника плит в геологии.** Геология дает интересный пример качественного структурного принципа; интересен он тем, что завоевал признание за последнее десятилетие, так что повышение его статуса еще свежо в памяти. Теория тектоники плит утверждает, что поверхность земного шара представляет собой набор огромных плит, общим числом несколько десятков, которые движутся с геологическими скоростями, сталкиваются, надвигаются или ныряют друг под друга, устремляясь к центру Земли и расплавляясь по мере их погружения. Движения плит объясняют форму и относительное расположение материков и океанов, районов вулканической и сейсмической активности, срединно-океанических хребтов и так далее. После уточнения дополнительных деталей, касающихся размеров и скоростей движения плит, теория в основном была построена.

Разумеется, она не была признана до тех пор, пока ей не удалось объяснить ряд фактов, связанных друг с другом и необъяснимых прежде, например соответствие флоры, фауны и «стратиграфических особенностей Западной Африки и северо-восточного побережья Южной Америки. Глобальная тектоника плит — это в высокой степени качественная теория. Теперь, когда она стала общепризнанной, кажется, что вся Земля предоставляет свидетельства в ее пользу повсеместно: ведь мы наконец описываем и видим Землю в соответствии с ее собственной структурой.

**Бактериальная теория инфекционных заболеваний.** Прошло немногим более ста лет с тех пор, когда Пастер предложил бактериальную теорию инфекций — качественный структурный принцип, вызвавший революцию в медицине. Эта теория постулирует, что большинство болезней возникает из-за присутствия и размножения в теле больного маленьких одноклеточных живых организмов, а заражение представляет собой перенос этих организмов от одного хозяина к другому. Значительная часть развития этой теории состоит в определении организмов, связанных с конкретными болезнями, их описании и выяснении их жизненного цикла. Тот факт, что из этого принципа есть много исключений (многие болезни не вызываются микробами), не умаляет его важности. Принцип требует, чтобы мы искали причину определенного типа; он не настаивает, что мы всегда найдем ее.

**Атомистическое учение.** Это учение составляет любопытный контраст тем только что описанным качественным структурным принципам. В том виде, в котором он возник из исследований Дальтона и его опытов, доказавших, что химические вещества соединяются в постоянных пропорциях, этот принцип дает типичный пример качественного описания структуры: элементы состоят из маленьких однородных частиц, одинаковых для одного элемента, но разных для разных элементов. Но поскольку лежащие в основе этого описания сорта атомов очень просты, а их разнообразие ограничено, вскоре были построены количественные теории, вобравшие в себя всю общую структуру исходной качественной гипотезы. В отношении же клеток, тектонических плит и бактерий соответствующие структуры столь богаты и многообразны, что исходные качественные принципы остаются несводимыми к количественным теориям, и их вклад в общую теорию ясно различим.

**Заключение:** Качественные структурные принципы встречаются в науке повсеместно. К их числу относятся многие из наших величайших научных открытий. Как показывают приведенные выше примеры, они часто задают рамки, в которых функционирует целая наука.

## ФИЗИЧЕСКИЕ СИМВОЛЬНЫЕ СИСТЕМЫ

Вернемся теперь к символам и определим *физическую символическую систему*. Прилагательное «физическая» означает две важные характеристики. Во-первых, такие системы явным образом подчиняются законам физики — они могут быть реализованы инженерными системами, сконструированными из технически реализуемых элементов. Во-вторых, хотя примененный нами термин «символ» предвосхищает ту его интерпретацию, которую мы имеем в виду, но он относится не только к символическим системам, используемым людьми.

Физическая символическая система состоит из набора элементов, называемых символами, которые представляют собой физические конфигурации, входящие в качестве компонент в элементы другого типа, называемые выражениями (или символическими структурами). Так, символическая структура состоит из некоторого числа позиций, занятых символами и связанных между собой некоторыми физическими соотношениями (например, одна позиция находится рядом с предыдущей). В любой момент времени система содержит некоторый набор таких символических структур. Кроме этих структур, система также содержит набор процессов, действующих на выражения и порождающих новые выражения: процессы создания, модификации, воспроизведения и уничтожения. Физическая символическая система — это машина, порождающая развивающийся во времени набор символических структур. Эта система существует в мире объектов, более широком, чем сами эти символические выражения.

Два понятия являются важнейшими для этой структуры, состоящей из выражений, символов и объектов: обозначение и интерпретация.

**Обозначение.** Выражение обозначает объект, если система, располагая этим выражением, может либо воздействовать на указанный объект, либо ее поведение зависит от этого объекта. В обоих случаях доступ к объекту достигается через соответствующее выражение, что и составляет суть обозначения.

**Интерпретация.** Система может интерпретировать выражение, если оно обозначает процесс и если, располагая этим выражением, система может осуществить указанный процесс.

Интерпретация предполагает особую форму зависимого действия: по данному выражению система может выполнить указанный процесс, другими словами, вызвать и исполнить свои собственные процессы исходя из выражений, которые эти процессы обозначают.



Система, способная к обозначению и интерпретации в вышеуказанном смысле, должна также удовлетворять ряду дополнительных требований полноты и замкнутости. Мы можем здесь лишь кратко их упомянуть; все они важны и имеют далекие идущие последствия.

1) Символ может использоваться для обозначения любого выражения. Тем самым, если у нас есть символ, заранее неизвестно, какие выражения он может обозначать. Эта произвольность относится только к символам; позиции для символов и их взаимоотношения определяют, какой объект обозначается сложным выражением. 2) Для каждого процесса, который может быть осуществлен машиной, существует по крайней мере одно выражение, обозначающее этот процесс. 3) Существуют процессы для создания любого выражения и для модификации любого выражения произвольным образом. 4) Выражения устойчивы: будучи однажды созданы, они продолжают существовать, пока не будут явным образом модифицированы или уничтожены. 5) Число выражений, которые система может хранить и использовать, по существу неограниченно.

Тот тип системы, который мы только что определили, нельзя считать незнакомым специалистам по информатике. Он сильно напоминает все универсальные компьютеры. Если для определения машины применить язык манипулирования символами, например Лисп, то это родство станет столь тесным, что нашу гипотетическую машину придется признать родной сестрой универсального компьютера. Мы описали подобную систему не для того, чтобы предложить что-то новое. Как раз наоборот: мы хотим показать, что сейчас известно или предполагается относительно систем и удовлетворяет такому описанию.

Теперь мы можем сформулировать научную гипотезу общего характера — качественный структурный принцип для символьных систем:

*Гипотеза о физической символьной системе.* Физическая символьная система обладает необходимыми и достаточными средствами для интеллектуального поведения общего характера.

Необходимость означает, что любая система, обладающая признаками универсального интеллекта, при ее анализе окажется физической символьной системой. Достаточность означает, что любая физическая символьная система достаточного размера может быть организована таким образом, чтобы проявлять интеллектуальное поведение общего характера. Формулировкой «интеллектуальное поведение общего характера» мы хотим указать на тот же диапазон применимости интеллекта,

который мы наблюдаем в поведении человека: что в любой реальной ситуации поведение соответствует целям системы и адаптивно к требованиям, предъявляемым окружающим миром, в пределах некоторых ограничений, относящихся к скорости и сложности.

Ясно, что гипотеза о физической символьной системе является качественным структурным принципом. Она характеризует общий класс систем, к которому относятся системы, способные к интеллектуальному поведению.

Этот принцип является эмпирической гипотезой. Мы определили некоторый класс систем; мы спрашиваем, объясняет ли этот класс ряд явлений, обнаруживаемых в реальном мире. Интеллектуальное поведение наблюдается повсюду вокруг нас в живой природе, в основном в поведении человека. Это та форма поведения, которую мы можем распознать по ее результатам независимо от того, осуществляется ли она людьми или же какими-то другими агентами. Эта гипотеза может на самом деле оказаться ошибочной. Интеллектуальное поведение не так просто осуществить, чтобы всякая система демонстрировала его вольно или невольно. В самом деле, некоторые люди в результате анализа приходят к выводу (по философским или научным соображениям), что эта гипотеза действительно ошибочна. С научной точки зрения, защищать или опровергать ее можно только одним способом: приводя эмпирические данные о реальном мире.

Теперь необходимо проследить развитие этой гипотезы и поискать опытные данные, могущие помочь ее проверить.

### РАЗВИТИЕ ГИПОТЕЗЫ О СИМВОЛЬНОЙ СИСТЕМЕ

Физическая символьная система является примером универсальной машины. Поэтому гипотеза о символьной системе предполагает, что интеллект должен быть реализован с помощью универсального компьютера. Однако эта гипотеза утверждает нечто существенно большее, чем рассуждение, часто выводимое из соображений физического детерминизма и состоящее в том, что любое реализуемое вычисление может быть реализовано универсальной машиной, если это вычисление задано. Эта гипотеза более конкретна: она утверждает, что интеллектуальная машина является символьной системой, и тем самым делает конкретное допущение об архитектуре и природе интеллектуальных систем. Важно понимать, откуда возникло это конкретное допущение.

**Формальная логика.** Корни этой гипотезы восходят к идеям Фреге, а также Рассела и Уайтхеда, которые стремились формализовать логику: реализовать фундаментальные математиче-

ские понятия в логике и тем самым поставить понятия доказательства и дедуктивного вывода на твердое основание. Эти усилия достигли кульминации в создании математической логики — знакомых нам пропозициональной логики, логики первого и высших порядков. Был выработан своеобразный подход, который часто называют «игрой в символы». Логика, в которую оказалась включенной вся математика, стала рассматриваться как игра ничего не обозначающими символами, ведущаяся по определенным, чисто синтаксическим правилам. Вся семантика была полностью изгнана из этой игры. Получилась механическая, хотя и довольно свободная (сейчас мы бы назвали ее недетерминистской) система, о которой можно было доказывать различные утверждения. Мы можем назвать это этапом формальных символьных манипуляций.

Этот общий подход хорошо отражен в развитии теории информации. Снова и снова повторялось, что Шеннон определил систему, полезную лишь для коммуникации и отбора и не имеющую никакого отношения к семантике. Выражались сожаления, что столь общее название присвоено такой узкой области, и были предприняты попытки переименовать ее в «теорию избирательной информации» — безуспешные, разумеется.

**Машины Тьюринга и цифровой компьютер.** Разработку первых цифровых компьютеров и теории автоматов, начиная с собственных исследований Тьюринга 30-х годов, можно рассмотреть одновременно. Они согласуются в подходе к тому, что считать существенным. Возьмем модель, разработанную самим Тьюрингом, поскольку на ней хорошо видны характерные черты этого подхода.

Машина Тьюринга содержит две памяти: неограниченную ленту и управляющий механизм с конечным числом состояний. Лента содержит данные, т. е. знаменитые нули и единицы. Набор операций, которые машина может совершать над лентой, очень мал: чтение, запись и просмотр. Операция чтения не меняет данных, но обеспечивает условные переходы управления в зависимости от данных, находящихся под читающей головкой. Как нам всем известно, эта модель содержит существенные свойства любого компьютера в смысле того, что компьютеры вообще могут делать, хотя разные компьютеры с разными регистрами памяти и разными наборами операций способны выполнять одни и те же вычисления, расходуя разные объемы памяти и требуя разного времени на их выполнение. В частности, модель, известная как машина Тьюринга, содержит в себе как понятие вычислимости (т. е. что может, а что не может быть вычислено), так и понятие универсальной машины, способной сделать все, что в принципе можно сделать на какой угодно машине.

Поразительно, что два самых глубоких открытия в обработке информации были сделаны в 30-х годах, прежде чем появились современные компьютеры. Этим мы обязаны гению Алана Тьюринга. Это также результат успехов математической логики того времени и свидетельство того, сколь многим информатика ей обязана. Одновременно с работами Тьюринга появились статьи логиков Эмиля Поста и (независимо) Алонсо Чёрча. Исходя из независимых понятий логистических систем (правил вывода Поста и рекурсивных функций соответственно), они пришли к аналогичным результатам о неразрешимости и универсальности — результатам, из которых, как вскоре было показано, следует, что все три упомянутые выше системы эквивалентны. В самом деле, тот факт, что все эти попытки определить наиболее широкий класс систем обработки информации привели независимо к сходным результатам, укрепляет нашу убежденность, что суть обработки информации адекватно выражена в этих моделях.

Ни в одной из этих систем на поверхности не видно понятия символа как элемента, который что-то обозначает. Данные рассматриваются просто как конечные последовательности из нулей и единиц; действительно, нейтральность данных существенна для сведения вычислений к физическим процессам. Система управления с конечным числом состояний всегда рассматривалась как небольшой по размерам автомат, и смысл логических игр заключался в том, чтобы увидеть, насколько малой может быть система состояний без разрушения универсальности машины. Насколько нам известно, никто никогда не играл в игры с динамическим добавлением новых состояний к управляющему устройству, т. е. не думал об управляющей памяти как о хранилище совокупности знаний системы. То, что было достигнуто на этом этапе, представляло собой половину принципа интерпретации: было показано, что машина может действовать исходя из описания. Этот этап можно назвать автоматическими формальными символьными манипуляциями.

**Понятие хранимой программы.** С разработкой второго поколения электронных машин в середине 40-х годов (после Эниака) появилось понятие хранимой программы. Это было справедливо названо поворотным пунктом в развитии информатики, как в концептуальном, так и в практическом отношении. Программы теперь стали данными, и с ними можно стало оперировать как с данными. Эта возможность, разумеется, уже присутствовала в неявном виде в модели Тьюринга: описания находятся на той же самой ленте, что и данные. Однако эта идея была реализована лишь тогда, когда машины обрели достаточно большую память, чтобы стало практичным размещать настоящие программы в каком-то внутреннем запоминающем

устройстве. В самом деле, у Эниака было всего 20 внутренних регистров памяти.

Концепция хранимой программы воплощает вторую половину принципа интерпретации, а именно ту его часть, в которой утверждается возможность интерпретации собственных данных системы. Но она еще не содержит понятия обозначения, т. е. физического соотношения, лежащего в основе семантики символов.

**Обработка списков.** Следующий шаг, сделанный в 1956 г., это обработка списков. Содержимым структур данных теперь были символы в том смысле, в каком они понимаются в нашей физической символьной системе: конфигурации, которые что-то обозначали, у которых были денотаты. Списки содержали адреса, через которые был возможен доступ к другим спискам; отсюда понятие списковых структур. То, что этот подход был новым, было неоднократно продемонстрировано нам в те первые дни обработки списков: коллеги спрашивали у нас, где же находятся данные, т. е. какой список в конце концов содержит тот набор бит, который является содержимым системы. Им казалось странным, что такого набора нет, а есть лишь символы, обозначающие другие символьные структуры.

Обработка списков означала сразу три момента в развитии информатики. Во-первых, это создание подлинно динамических структур памяти в машине, которая ранее считалась обладающей лишь фиксированной структурой. Это добавило к нашему набору операций такие, которые строили и модифицировали структуру, а не только заменяли и изменяли содержимое. Во-вторых, это стало первой демонстрацией фундаментальной концепции: компьютер состоит из набора типов данных и набора операций, подходящих для этих типов данных, так что вычислительная система должна использовать любые типы данных, удобные для решаемых ею задач, независимо от машины, на которой эти вычисления реализуются. В-третьих, обработка списков создала модель процедуры обозначения и тем самым определила символьную манипуляцию в том смысле, в котором мы употребляем это понятие в современной информатике.

Как это часто бывает, практика того времени уже предвосхищала все элементы обработки списков: адреса, само собой, использовались для обеспечения доступа к данным, в машинах с магнитными барабанами применялись записанные последовательно, одна за другой, программы (последовательная адресация) и так далее. Но понятие обработки списков в качестве абстракции создал новый мир, в котором обозначение и динамические символьные структуры стали определяющими характеристиками. Погружение первых систем обработки списков в языки (вроде Лиспа) часто описывается как препятствие

широкому внедрению методов обработки списков в практику программирования; на самом деле это было средством собрать воедино все элементы, составляющие новую абстракцию.

**Лисп.** Еще один шаг, заслуживающий упоминания, это создание в 1956—1960 гг. языка Лисп [2]. Оно завершило обобщение, освободив списковые структуры из плена, из погружения в конкретные машины, и создало новую формальную систему с S-выражениями, которая, как можно показать, эквивалентна другим универсальным схемам вычисления.

**Заключение.** То, что понятие значащего символа и символьной манипуляции не возникло вплоть до середины 50-х годов, вовсе не означает, что предшествующие этому шаги были не существенными или менее важными. В своей целокупности это понятие объединяет вычислимость, физическую реализуемость (посредством множества технологических приемов), универсальность, символьное представление процессов (т. е. интерпретируемость) и, наконец, символьные структуры и обозначение. Каждый из этих шагов представляет существенную часть целого.

Первый шаг в этой цепочке, осуществленный Тьюрингом, был мотивирован теоретически, но все остальные имеют глубокие эмпирические корни. Нас вело вперед развитие самих компьютеров. Принцип хранимой программы возник из опыта работы с Эниаком. Обработка списков выросла из попыток создать интеллектуальные программы. Она была подсказана появлением запоминающих устройств с произвольным доступом, обеспечившим четкую физическую реализацию обозначающего символа в виде адреса. Лисп стал итогом накопления практического опыта обработки списков.

### ЭМПИРИЧЕСКИЕ ДОВОДЫ

Перейдем теперь к эмпирическим доводам в поддержку гипотезы о том, что физические символьные системы способны к интеллектуальному поведению и что интеллектуальное поведение общего характера требует физической символьной системы. Эта гипотеза является эмпирическим обобщением, а не теоремой. Нам не известно никакого способа продемонстрировать связь между символьными системами и интеллектом чисто логическим путем. При отсутствии такой возможности остается обратиться к фактам. Наша главная цель, однако, не подробный обзор фактов, а использование доступного нам примера для иллюстрации утверждения о том, что информатика является эмпирической наукой. Поэтому мы лишь укажем, какого рода фактами мы располагаем и какова общая природа процесса опытной проверки.

Понятие физической символической системы приняло в основном современную форму в середине 50-х годов, и с этого времени можно проследить рост исследований в области искусственного интеллекта как единого направления в информатике. Двадцать лет последующей работы привели к накоплению фактических результатов, относящихся к двум основным темам. Первая группа результатов касается *достаточности* физических символических систем для порождения интеллекта; это попытки строить и проверять конкретные системы, обладающие такими возможностями. Вторая группа фактов обращается к *необходимости* иметь физическую символическую систему во всех случаях, когда имеются проявления интеллекта. Эти исследования начались с Человека, самой известной интеллектуальной системы, и попыток выяснить, можно ли объяснить его интеллектуальное поведение как работу физической символической системы. Есть и другие виды опытных данных, которые будут вкратце обсуждены ниже, но эти две группы являются важнейшими. Мы рассмотрим их по очереди. Первая имеет общее название искусственного интеллекта, вторая — исследований по когнитивной психологии.

**Конструирование интеллектуальных систем.** Основная парадигма первоначальной проверки бактериальной теории заболевания была следующей: выявите болезнь; затем ищите микроб. Аналогичная парадигма инспирировала значительную часть исследований в области искусственного интеллекта: выявите класс задач, решение которых требует интеллектуальных способностей; затем разработайте программу для цифрового компьютера, способную решать такие задачи. Простые и хорошо структурированные области первыми привлекли внимание: игры и головоломки, задачи исследования операций типа составления расписаний и распределения ресурсов, простые задачи на индукцию. Дюжины, если не сотни программ такого рода написаны к настоящему времени, каждая из которых в той или иной степени способна к интеллектуальному поведению в соответствующей области. Конечно, интеллектуальность — это не такая вещь, которая либо присутствует в полном объеме, либо отсутствует целиком, и постоянно происходило развитие к более высоким уровням мастерства в конкретных предметных областях и к расширению самих этих областей. Первые шахматные программы считались успешными, если они могли соблюдать правила игры и проявляли некие признаки целесообразности поведения; чуть позже они достигли уровня начинающих игроков-людей; еще через 10—15 лет они стали выигрывать у серьезных любителей. Прогресс был медленным (а общие трудовые затраты программистов — незначительными), но постоянным, и парадигма «построить и испытать» пре-

ходила свой обычный цикл: в целом исследовательская активность воспроизводила на макроскопическом уровне базисный цикл «породить и проверить» многих программ искусственного интеллекта.

Существует постоянно расширяющаяся область, в которой удается достичь интеллектуального поведения. От исходных задач исследователи перешли к более общим и трудным: к системам для чтения, понимания и генерации текстов на естественном языке самыми разными способами, к системам интерпретации визуальных сцен, к системам координации движений руки и глаза, системам проектирования, системам, пишущим компьютерные программы, системам распознавания устной речи — этот список, если не бесконечен, то очень длинен по крайней мере. И если существуют какие-то ограничения, за пределами которых наша гипотеза не работает, то пока что таких границ не видно. Вплоть до сегодняшнего дня скорость прогресса в основном определялась сравнительно скромными ресурсами, отводимыми на научные исследования в этой области, и неизбежностью затраты серьезных усилий и средств на каждый крупный новый проект по созданию системы.

Происходило, конечно, и много другого, чем просто накопление примеров интеллектуальных систем, приспособленных к решению задач из конкретных предметных областей. Было бы удивительным и непривлекательным, если бы оказалось, что программы искусственного интеллекта, выполняющие эти разнообразные функции, не имеют между собой ничего общего, кроме того, что это все примеры физических символьных систем. Поэтому существовал большой интерес к поискам механизмов, обладающих общностью, и общих компонент среди программ, выполняющих разные задачи. Этот поиск выводит теорию за рамки первоначальной гипотезы о символьных системах к более полному описанию особого рода символьных систем, эффективных в области искусственного интеллекта. Во втором разделе нашей работы мы обсудим один пример гипотезы этого второго уровня специфичности: гипотезу эвристического поиска.

Поиск общности стимулировал создание нескольких программ, разработанных для вычленения общих механизмов решения задач, их отделения от требований, предъявляемых конкретными предметными областями. Общий решатель задач (General Problem Solver, GPS) был, вероятно, первым из них; среди его потомков есть такие современные системы, как PLANNER и CONNIVER. Поиск общих компонент привел к общим схемам представления целей и планов, методам построения дискриминирующих сетей, процедурам управления поиском на дереве, механизмам сопоставления с образцом и системам



грамматического разбора. В настоящее время ведется экспериментальная работа с целью найти удобные средства представления временных соотношений предшествования и последовательности действий, причинности, движения и других подобных грамматических и логических категорий. Во все большей степени становится возможным собирать большие интеллектуальные системы из таких базовых компонент как из модулей.

Можно получить некоторое общее представление о том, что происходит, опять обратившись к аналогии с бактериальной теорией болезней. Если первый всплеск исследований, связанных с этой теорией, в основном состоял в поиске микроорганизмов, связанных с каждой болезнью, то последующие усилия были направлены на изучение того, что представляет собой микроб, т. е. на построение нового структурного уровня на фундаменте исходного качественного структурного принципа. В области искусственного интеллекта ранние исследования имели целью построение интеллектуальных программ для широкого многообразия почти случайно выбранных задач, но в дальнейшем исследования сконцентрировались на более избирательно намечаемых целях в стремлении найти и понять общие механизмы для таких систем.

**Моделирование использования символов человеком.** Гипотеза о символьных системах предполагает, что причиной символьного поведения человека служит то, что он обладает свойствами физической символьной системы. Поэтому результаты попыток смоделировать поведение человека символьными системами становятся важной частью свидетельств в поддержку этой гипотезы, и исследования по искусственному интеллекту проводятся в тесном сотрудничестве с психологией обработки информации, как обычно называют эту область.

Поиск объяснений интеллектуального поведения человека в рамках символьных систем увенчался крупными достижениями за последние 20 лет, вплоть до того, что теория обработки информации стала ведущим современным подходом в когнитивной психологии (психологии интеллекта). В особенности в таких областях, как решение задач, формирование понятий и долговременная память, модели символьной манипуляции играют ныне решающую роль.

Исследования психологии обработки информации включают два основных вида эмпирического поиска. Первый из них — проведение наблюдений и постановка экспериментов по поведению человека в ходе выполнения заданий, требующих интеллектуальной активности. Второй, очень похожий на параллельные исследования по искусственному интеллекту, — программирование символьных систем для имитации наблюдаемого поведения человека. Психологические наблюдения и экспери-

менты привели к формулированию гипотез о символических процессах, используемых людьми, и эти гипотезы служат важными источниками идей, пригодных для построения программ. Так, многие идеи основных механизмов GPS были выведены из тщательного изучения протоколов опытов, в которых испытуемые размышляли вслух, выполняя задания по решению задач.

Эмпирический характер информатики теперь стал еще более очевиден, чем показывает эта связь с психологией. Дело не только в том, что для проверки реалистичности объяснений человеческого поведения имитационными моделями необходимы психологические эксперименты, но и в том, что из экспериментов выводятся новые идеи проектирования и построения физических символических систем.

**Другие свидетельства.** Основной корпус данных в пользу гипотезы о физических символических системах, которые мы до сих пор не рассматривали, это отрицательные данные, т. е. отсутствие конкретных альтернативных гипотез о том, как может быть реализовано интеллектуальное поведение — безразлично, человека или машины. Большая часть попыток построить такие гипотезы предпринималась в области психологии. Здесь имеется непрерывный спектр теорий, на одном полюсе которого лежат точки зрения, обычно называемые «бихейвиоризмом», а на другом — «гештальт-психологией». Ни одна из этих точек зрения не может быть реальным конкурентом гипотезы о физических символических системах по двум следующим причинам. Во-первых, ни бихейвиористы, ни «гештальтисты» до сих пор не показали и даже не предложили, как можно было бы показать, что постулируемые ими механизмы достаточны для объяснения интеллектуального поведения при выполнении сложных заданий. Во-вторых, ни одна из этих теорий не сформулирована со степенью конкретности, хотя бы отдаленно напоминающей конкретность программ искусственного интеллекта. Фактически альтернативные теории достаточно расплывчаты для того, чтобы их интерпретация в духе теории обработки информации не была чрезмерно трудным делом; это позволило бы включить их в гипотезу о физических символических системах как частные случаи.

## ЗАКЛЮЧЕНИЕ

Мы привели пример гипотезы о физических символических системах для того, чтобы дать конкретную иллюстрацию основного тезиса. Тезис этот утверждает, что информатика является научной дисциплиной в обычном смысле этого термина: она выдвигает научные гипотезы и затем ищет их подтверждения путем экспериментирования. Есть, однако, и другая причина:

для выбора именно этой гипотезы для иллюстрации нашего тезиса. Гипотеза о физических символьных системах сама по себе является важной научной гипотезой того сорта, который мы ранее назвали «качественными структурными принципами». Она представляет собой крупное открытие в области информатики, которое, если его удастся подтвердить экспериментально, что, по-видимому, и произойдет, окажет существенное и длительное воздействие на эту науку.

Теперь мы обратимся ко второму примеру, роли поиска в интеллектуальной деятельности. Эта тема и частная гипотеза о роли поиска, которую мы рассмотрим, также были очень важны для информатики в целом и для искусственного интеллекта в особенности.

## 2. ЭВРИСТИЧЕСКИЙ ПОИСК

Знание о том, что физические символьные системы обеспечивают возможность интеллектуального поведения, не говорит нам ничего о том, как они этого добиваются. Наш второй пример качественного структурного принципа в информатике отвечает на этот второй вопрос, утверждая, что символьные системы решают задачи, используя процесс эвристического поиска. Это общее положение, как и предыдущее, основано на опытных данных, а не выведено формально из каких-то других посылок. Однако мы скоро увидим, что оно на самом деле имеет некоторую логическую связь с гипотезой о символьных системах, и, вероятно, можно предвидеть формализацию этой связи в будущем. Пока это время не пришло, нам снова придется рассказывать об эмпирическом исследовании. Мы опишем, что известно о самом эвристическом поиске и подытожим опытные факты, показывающие, как он обеспечивает разумность действий. Мы начнем с формулировки этого качественного структурного принципа — гипотезы об эвристическом поиске.

*Гипотеза об эвристическом поиске.* Решения задач представляются в виде символьных структур. Физическая символьная система проявляет свою интеллектуальность при решении задач посредством поиска — т. е. порождением и последовательной модификацией символьных структур, пока этот процесс не приведет к структуре, отвечающей решению.

Физические символьные системы должны использовать эвристический поиск при решении задач в силу того обстоятельства, что вычислительные возможности таких систем ограничены: за конечное число шагов и в течение ограниченного периода времени они способны осуществить лишь конечное число процессов. Разумеется, это ограничение не очень сильное, так как

оно распространяется на все универсальные машины Тьюринга. Мы, однако, имеем в виду более сильное ограничение — практическую ограниченность. Можно представить себе системы, неограниченные в практическом смысле, а способные, например, вести поиск, параллельно просматривая вершины экспоненциально ветвящегося дерева с постоянной скоростью продвижения в глубину. Нас здесь будут интересовать не такие системы, а те, вычислительные ресурсы которых малы по сравнению со сложностью ситуаций, с которыми они сталкиваются. Такое ограничение не исключает никаких реально существующих символьных систем, идет ли речь о людях или о компьютерах, если реальны сами задачи. Ограниченность ресурсов позволяет нам в большинстве случаев рассматривать символьную систему, как если бы она была сериальным устройством, способным в каждый момент времени выполнять лишь один процесс. Если она способна выполнить лишь небольшой объем вычислений в любой короткий интервал времени, то мы вполне можем считать, что она производит вычисления последовательно. Поэтому «символьная система с ограниченными ресурсами» и «сериальная символьная система» — это практически синонимы. Проблема распределения ограниченных ресурсов по интервалам времени обычно может рассматриваться, если эти интервалы достаточно короткие, как проблема организации работы сериальной машины.

### РЕШЕНИЕ ЗАДАЧ

Поскольку способность решать задачи обычно считается главным признаком интеллектуальности системы, то естественно, что большая часть истории искусственного интеллекта занимают попытки построить и понять системы решения задач. Вот уже два тысячелетия философы и психологи обсуждают решение задач, и их рассуждения насквозь пронизаны ощущением тайны. Если вы думаете, что нет ничего загадочного и таинственного в символьной системе, решающей задачи, то вы дитя нашего времени, чьи взгляды сформировались не раньше середины века. Платон (и, согласно его рассказу, Сократ) считали трудным для понимания даже то, как задачи могут быть поставлены, не говоря уже о том, как они могут быть решены. Позвольте вам напомнить, как он формулирует эту загадку в диалоге «Менон»:

*Менон:* Но каким же образом, Сократ, ты будешь искать вещь, не зная даже, что она такое? Какую из неизвестных тебе вещей изберешь ты предметом исследования? И если ты в лучшем случае натолкнешься на нее, откуда ты узнаешь, что она — именно то, чего ты не знал?

Чтобы справиться с этой головоломкой, Платон изобрел свою знаменитую теорию припоминания: когда вы думаете, что вы открываете или изобретаете что-то, вы на самом деле лишь припоминаете то, что вы уже знали в предшествующем существовании. Если это объяснение кажется вам слишком нелепым, то в наше время есть и другое, значительно более простое и основанное на понимании того, что такое символичные системы. Приблизительно его можно изложить так:

Поставить задачу — это обозначить (1) *тест* для некоторого класса символических структур (решений задачи) и (2) *генератор* символических структур (потенциальных решений). Решить задачу — это породить структуру с помощью (2), которая удовлетворяет тесту (1).

Перед нами стоит задача, если мы знаем, что мы хотим сделать (имеется тест), и при этом мы не знаем немедленно, как это сделать (наш генератор не порождает немедленно символическую структуру, удовлетворяющую тесту). Символическая система способна ставить и решать задачи (иногда), потому что она может порождать и проверять символические структуры.

Если бы этим исчерпывалось все, что нужно для решения задач, то почему бы просто не породить сразу именно такое выражение, которое удовлетворяет тесту? Фактически именно это мы и делаем, когда желаем и мечтаем. «Если бы желания были лошадьми, то бродяги могли бы ездить верхом» — гласит пословица. Но за пределами мира фантазий это невозможно. Знать, как мы стали бы проверять нечто уже построенное, — это не то же самое, что знать, как это построить, т. е. это не означает, что у нас есть какой-либо генератор, способный порождать решения.

Например, все знают, что значит «решить» задачу, если наша цель — одержать победу в шахматной партии. Существует простой тест для определения выигрышных позиций, а именно: король противника не может избежать шаха, т. е. ему поставлен мат. В мире фантазий мы просто порождаем стратегию, которая ведет к мату при любых контрстратегиях противника. Увы, неизвестно никакого генератора подобной стратегии для существующих символических систем (людей или машин). Вместо этого хорошие ходы в шахматной игре ищутся путем порождения различных вариантов и последующей мучительной процедуры их оценивания с использованием приближенных и часто ошибочных мер, которые предположительно указывают вероятность того, что та или иная последовательность ходов лежит на маршруте, приводящем к выигрышной позиции. Генераторы ходов существуют, не существует генераторов выигрышных ходов.

Прежде чем для данной задачи можно будет создать генератор ходов, должно существовать проблемное пространство, т. е. пространство символьных структур, в котором можно представить проблемные ситуации, включая исходную и целевую. Генераторы ходов — это процессы модификации одной ситуации из проблемного пространства в другую ситуацию того же пространства. Основные характеристики физических символьных систем гарантируют, что они могут представлять проблемные пространства и обладают генераторами ходов. Каким образом в каждой конкретной ситуации они синтезируют проблемное пространство и генераторы ходов, подходящие для этой ситуации, — это вопрос, все еще в очень большой степени находящийся на переднем крае исследований по искусственному интеллекту.

Когда символьная система располагает постановкой задачи и проблемным пространством, она должна выполнить следующее задание: использовать свои ограниченные вычислительные ресурсы для порождения возможных решений, одного за другим, пока она не найдет такое, которое удовлетворяет тесту, определяющему постановку задачи. Если бы система могла как-то управлять порядком, в котором порождаются потенциальные решения, то было бы желательно, чтобы настоящие решения обладали высокой вероятностью быть порожденными на ранних этапах этого процесса. Символьная система проявляет интеллектуальность в той степени, в которой она преуспела в этом. Для системы с ограниченными вычислительными ресурсами интеллектуальность состоит в том, чтобы на каждом этапе мудро выбирать следующий шаг.

### ПОИСК В РЕШЕНИИ ЗАДАЧ

В течение первых десяти лет исследований по искусственному интеллекту изучение решения задач означало почти то же самое, что изучение процессов поиска. Из наших описаний сущности задач и решения задач легко видеть, почему это было так. На самом деле можно было бы спросить, могло ли быть по-другому. Но прежде чем отвечать на этот вопрос, мы должны глубже изучить природу процессов поиска, насколько она выявилась за эти десять лет исследований.

**Извлечение информации из проблемного пространства.** Рассмотрим набор символьных структур, небольшое подмножество которого составляют решения данной задачи. Предположим кроме того, что решения случайным образом распределены по всему пространству. Это означает, что нет никакой информации, которая позволила бы любому генератору поиска работать лучше, чем случайный поиск. Тогда никакая символъ-

ная система не смогла бы проявить большую (или меньшую) интеллектуальность, чем другая, при решении задачи, хотя одной могло бы повезти больше, чем другой.

Поэтому необходимое условие появления интеллекта состоит в том, что распределение решений не является полностью случайным, т. е. пространство символьных структур обладает по крайней мере некоторой степенью организованности и упорядоченности. Второе условие — то, что этот порядок в пространстве символьных структур является более или менее распознаваемым. Третье условие интеллектуальности — то, что генератор потенциальных решений способен вести себя по-разному в зависимости от того, какой именно порядок обнаружен. Проблемное пространство должно содержать информацию, а символьная система должна быть способна извлекать и использовать ее. Для начала рассмотрим очень простой пример, в котором интеллектуальность очень просто обеспечить.

Рассмотрим задачу разрешения простого алгебраического уравнения

$$AX+B=CX+D.$$

Тест определяет решение как любое выражение вида  $X=E$ , такое что  $AE+B=CE+D$ . Теперь можно использовать в качестве генератора любой процесс, порождающий числа, которые можно было проверить подстановкой в последнее уравнение. Мы не можем называть такой генератор интеллектуальным.

Другой способ — использовать генераторы, опирающиеся на тот факт, что исходное уравнение можно модифицировать добавлением или вычитанием равных величин из обеих частей уравнения, а также умножением или делением их на равные величины, не изменяя его решений. Но конечно, мы можем получить еще больше информации для управления генератором, сравнив исходное выражение с видом решения и сделав именно те изменения в уравнении, которые оставляют его решение неизменным и одновременно приводят его к желаемому виду. Такой генератор может заметить, что в правой части исходного уравнения имеется нежелательный член  $CX$ , вычесть его из обеих частей и снова сгруппировать однородные члены. Затем он может заметить нежелательный член  $B$  в левой части и вычесть его. Наконец, он может избавиться от нежелательного коэффициента  $(A-C)$  в левой части посредством деления.

Так в ходе этой процедуры, которая теперь проявляет явные черты интеллектуальности, генератор порождает последовательно сменяющие друг друга символьные структуры, каждая из которых получается модификацией предыдущей; эти модификации нацелены на уменьшение различий между исходной структурой и формой выражения, являющегося тестом; при

этом другие условия, которым должно отвечать решение, сохраняются.

Этот простой пример уже иллюстрирует многие основные механизмы, применяемые символьными системами при интеллектуальном решении задач. Во-первых, каждое последующее выражение не порождается независимо, но порождается модификацией порожденного ранее. Во-вторых, эти модификации делаются не наугад, а зависят от информации двух разных типов. Они зависят от информации, постоянной для всего этого класса алгебраических задач, и поэтому встроенной в саму структуру генератора: все модификации выражений должны оставлять решения уравнения неизменными. Модификации также зависят от информации, изменяющейся на каждом шаге: обнаруженных различий в форме, которые еще остаются, между текущим выражением и желаемым выражением. В результате генератор включает в себя некоторые тесты, которым должно удовлетворять решение: выражения, не удовлетворяющие этим тестам, никогда не будут порождены. Использование информации первого типа гарантирует, что действительно порождается лишь небольшое подмножество всех возможных выражений, но это подмножество содержит решение. Использование информации второго типа позволяет достичь решения посредством последовательных приближений, применяя простую форму анализа соответствия средств и целей для направления поиска.

Нет никакой тайны в том, откуда берется информация, направляющая поиск. Нам не нужно следовать Платону, наделяя символьные системы предыдущим существованием, в котором они уже знали решения. Умеренно сложная система порождения и проверки решает задачу без всякой нужды в теории переселения душ.

**Деревья поиска.** Простая алгебраическая задача может показаться необычным, даже патологическим примером поиска. Это несомненно не метод проб и ошибок, так как хотя в нем было несколько проб, но не было ни одной ошибки. Мы больше привыкли думать о поиске при решении задач как о порождении обильно ветвящихся деревьев возможных частичных решений, число которых может доходить до тысяч, даже миллионов ветвей, прежде чем одна из них приведет к решению. Так, если из каждого порожденного генератором выражения он затем порождает  $B$  новых ветвей, то дерево будет расти как  $B^D$ , где  $D$  — глубина. Дерево, выросшее для алгебраической задачи, обладает тем странным свойством, что его коэффициент ветвления  $B$  равен единице.

Шахматные программы обычно порождают широкие деревья поиска, в некоторых случаях достигающие миллиона ветвей и



более. (Хотя этот пример помогает проиллюстрировать наши утверждения о деревьях поиска, необходимо заметить, что цель поиска в шахматах — не порождение возможных решений, а их оценивание (проверка).) Одна из тем исследований игровых программ в основном была связана с улучшением представления шахматной доски и процессов ходов на этой доске с целью ускорить поиск и сделать возможным поиск на деревьях больших размеров. Мотивация этого направления, конечно, та, что чем глубже динамический поиск, тем точнее будут оценки, которые его завершают. С другой стороны, есть надежные эмпирические данные о том, что сильнейшие шахматисты-люди, т. е. гроссмейстеры, редко изучают деревья, содержащие более ста вариантов. Эта экономия достигается не столько за счет меньшей по сравнению с шахматными программами глубины поиска, сколько за счет очень редкого и избирательного ветвления в каждой вершине. Без ухудшения качества оценок это возможно лишь при встраивании большей избирательности в сам генератор, чтобы он мог выбирать для порождения лишь те ветви, которые с очень высокой вероятностью дадут важную и нужную информацию о качестве позиции.

До некоторой степени парадоксально звучащий вывод, к которому привело это обсуждение, состоит в том, что поиск, т. е. последовательное порождение структур, потенциально способных представлять решение, является основополагающим аспектом проявления символьными системами интеллектуальности при решении задач, но объем поиска не является мерой интеллектуальности, при этом проявленной. Задачу делает трудной не большой объем поиска, необходимого для отыскания ее решения, а большой объем поиска, который оказался бы необходимым, если бы не был применен требуемый уровень интеллектуальности. Когда символьная система, пытающаяся решить задачу, знает достаточно о том, что ей делать, она прямо идет к своей цели; но всякий раз, когда ее знание становится недостаточным, когда она вступает на неведомую землю, она сталкивается с угрозой пройти через обширный поиск, прежде чем снова найти свой путь.

Возможность экспоненциального взрывного роста дерева поиска, присущая любой схеме порождения решений задачи, предупреждает нас, что не стоит полагаться на грубую силу компьютеров — даже самых больших и быстрых — в качестве компенсации невежества и неразборчивости генераторов решений. Время от времени у некоторых людей возникает надежда, что будет найден достаточно быстродействующий компьютер, который можно будет так хитроумно запрограммировать, чтобы он смог хорошо играть в шахматы методом слепого поиска. О шахматной игре неизвестно ничего такого, что исключало бы

такую возможность теоретически. Эмпирические исследования управления поиском по деревьям больших размеров, приведшие к довольно скромным результатам, сделали это направление намного менее перспективным, чем оно было тогда, когда шахматы были впервые выбраны в качестве подходящего объекта исследований по искусственному интеллекту. Мы должны считать этот факт одним из важных эмпирических результатов изучения шахматных программ.

**Формы интеллектуальности.** Назначение интеллекта тем самым состоит в том, чтобы предотвратить всегда возможный экспоненциальный рост дерева поиска (комбинаторный взрыв). Как этого можно добиться? Первый способ, уже проиллюстрированный примером решения алгебраического уравнения и шахматными программами, генерирующими только «хорошие» ходы для последующего анализа, это встроить избирательность в генератор: порождать только такие структуры, которые имеют шанс оказаться решениями или лежать на пути, ведущем к решению. Обычно это приводит к уменьшению ветвления, но не устраняет его полностью. Взрывной экспоненциальный рост в конечном итоге не предотвращается (за исключением особенно хорошо структурированных ситуаций, как в примере из алгебры), а лишь отодвигается на некоторый срок. Поэтому интеллектуальная система должна дополнить избирательность своего генератора решений какими-то другими методами направления поиска, использующими информацию о структуре пространства решений.

Двадцатилетний опыт управления поиском по дереву в разных проблемных ситуациях дал небольшой набор общих методов, составляющий часть инструментария любого исследователя искусственного интеллекта. Поскольку эти методы были описаны в общих обзорах вроде [4], мы можем очень кратко подытожить их здесь.

В последовательном эвристическом поиске основной вопрос всегда стоит так: что делать дальше? В поиске по дереву этот вопрос в свою очередь разбивается на два вопроса: [1] из какой вершины дерева вести поиск дальше? [2] в каком направлении двигаться из этой вершины? Информация, полезная для ответа на первый из этих вопросов, может быть рассматриваема как мера относительного расстояния различных вершин от цели. Информация, полезная для ответа на второй вопрос (в каком направлении вести поиск), часто может быть получена, как в алгебраическом примере, обнаружением специфических различий между структурой, представленной текущей вершиной, и структурой, обозначающей цель и описываемой тестом, с последующим выбором действий, связанных с уменьшением этих специфических различий. В этом и состоит метод, из-

вестный как анализ средств и целей; он играет центральную роль в структуре GPS (общего решателя проблем).

Важность эмпирических работ в качестве источника общих идей в исследованиях по ИИ можно ясно продемонстрировать, проследив историю создания многочисленных программ для решения задач, в которой центральную роль играли две эти идеи: поиск с выбором первыми самых лучших кандидатов и анализ средств и целей. Зачатки выбора первыми самых перспективных промежуточных решений уже присутствовали, хотя сам метод тогда еще не имел названия, в программе «Logic Theorist» 1955 г. Программа GPS, воплощающая анализ средств и целей, появилась около 1957 г., но она совмещала этот анализ не с выбором первой лучшей вершины, а с модифицированным методом поиска в глубину. Шахматные программы как правило были связаны (по соображениям экономики памяти) с поиском в глубину, дополненным после 1958 г. мощной процедурой альфа-бета усечения. Похоже, что каждый из этих методов неоднократно изобретался заново, и трудно найти общие, не зависящие от конкретной задачи теоретические обсуждения решения задач в терминах этих концепций до середины или второй половины 60-х годов. Масштабы поддержки, которую они получили со стороны формализованной математической теории, все еще незначительны: некоторые теоремы о сокращении объема поиска, которое может быть обеспечено применением альфа-бета эвристики, пара теорем (рассмотренных в [4]) о поиске по кратчайшему пути, и несколько полученных в самое последнее время теорем о поиске с выбором лучшей вершины с применением вероятностной оценочной функции.

**«Слабые» и «сильные» методы.** Рассмотренные до сих пор методы ставили своей целью ограничение, а не предотвращение экспоненциального роста. По этой причине они были удачно названы «слабыми методами» — методами, которые следует применять, когда знания символической системы или степень структурированности проблемного пространства недостаточны для того, чтобы полностью исключить поиск. Поучительно сравнить высокоструктурированную ситуацию, которая может быть выражена, скажем, как задача линейного программирования, с менее структурированными ситуациями комбинаторных задач вроде задачи о коммивояжере или составления расписаний. («Менее структурированная» здесь означает недостаточность или отсутствие теории, описывающей структуру проблемного пространства.)

При решении задач линейного программирования может потребоваться большой объем вычислений, но поиск не ветвится. Каждый шаг — это шаг на пути к решению. При решении комбинаторных задач или при доказательстве теорем поиска по

дереву редко удается избежать, и успех зависит от эвристических методов поиска описанного выше типа.

Не все направления решения задач в области ИИ следовали обрисованному выше пути. Пример несколько иного подхода дают работы по системам доказательства теорем. Здесь идеи, заимствованные из математики и логики, сильно повлияли на направление исследований. Например, использование эвристик встречало сопротивление в тех случаях, когда полнота не могла быть доказана (что несколько странно, так как про наиболее интересные математические системы известно, что они неразрешимы). Поскольку полнота редко может быть доказана для эвристических методов поиска с выбором первой лучшей вершины или для многих видов селективных генераторов, то результат этого требования был весьма разочаровывающим. Когда программы доказательства теорем стали постоянно терпеть неудачи из-за комбинаторных взрывов их деревьев поиска, разработчики обратились к селективным эвристикам, которые во многих случаях оказались аналогичными эвристикам, используемым в общих программах решения задач. Например, эвристика множества поддержки — это разновидность обратного поиска, приспособленного к проблемной среде доказательства теорем методом разрешения.

**Итоги опыта работы.** Мы описали, как действует наш второй качественный структурный принцип, утверждающий, что символьные системы решают задачи посредством эвристического поиска. Кроме того, мы изучили некоторые дополнительные свойства эвристического поиска, в частности, угрозу, что он всегда может столкнуться с экспоненциальным ростом дерева поиска, а также некоторые средства, которые он использует для борьбы с этой угрозой. По поводу того, насколько эвристический поиск был эффективен в качестве механизма решения задач, мнения различны и зависят от того, какие проблемные области рассматривались и какие критерии адекватности принимались. Успех гарантирован, если мы не слишком многого хотим, а если хотеть слишком многого, то гарантирован провал. Результаты практической работы с такими программами можно подытожить примерно так. Немногие программы решают задачи на профессиональном уровне экспертов. Наиболее известные примеры «экспертных» систем — программа для игры в шашки Сэмюэла и система DENDRAL Фейгенбаума и Ледерберга, но можно указать также многие программы эвристического поиска для таких проблемных областей, как составление расписаний и целочисленное программирование в исследовании операций. В ряде областей программы действуют на уровне компетентных непрофессионалов: шахматы, некоторые проблемные области доказательства теорем, многие

виды игр и головоломок. Уровень компетентности, свойственный людям, пока что недоступен программам, имеющим сложный перцептивный «вход»: распознавателям устной речи, анализаторам визуальных сцен, роботам, передвигающимся в реальном пространстве и времени. Тем не менее и в этих сложных проблемных ситуациях достигнуты впечатляющие успехи и накоплен большой опыт.

Пока что мы не располагаем глубокими теоретическими объяснениями, почему возникает именно такая картина качества работы программ для разных областей. Из эмпирических данных, однако, можно вывести два утверждения. Во-первых, из того, что нам известно о поведении людей-экспертов при выполнении ими заданий вроде игры в шахматы, можно с большой вероятностью предположить, что любая система, способная достичь такого уровня компетентности, должна иметь доступ к очень большим объемам семантической информации, хранимым в ее памяти. Во-вторых, превосходство человека в заданиях, связанных с большими объемами перцептивной информации, отчасти может быть приписано специализированным встроенным структурам параллельной обработки информации, присущим уху и глазу человека.

В любом случае качество работы систем должно обязательно зависеть и от свойств проблемных областей, и от свойств символьных систем, используемых для решения задач из этих областей. Для большинства интересующих нас областей реальной жизни структура пространства решений оказалась недостаточно простой, чтобы дать какие-либо теоремы о сложности или как-то иначе, нежели эмпирически, подсказать, насколько велики задачи реального мира по сравнению с возможностями наших символьных систем их решать. Эта ситуация может измениться, но пока это не произошло, мы должны полагаться на эмпирические исследования, применяя лучшие решатели задач, которые мы можем построить, в качестве основного источника знаний о масштабах и особенностях сложности задач. Даже в высокоструктурированных областях вроде линейного программирования теория была намного полезнее в укреплении эвристических подходов, лежащих в основе наиболее мощных алгоритмов решения таких задач, чем в углублении анализа их сложности.

### ИНТЕЛЛЕКТ БЕЗ ОБШИРНОГО ПОИСКА

Наш анализ интеллектуальности приравнял ее к способности извлекать и использовать информацию о структуре проблемного пространства с целью сделать порождение решения задачи настолько быстрым и непосредственным, насколько это возмож-

но. Новые направления улучшения способности символьных систем решать задачи можно отождествить тем самым с новыми способами извлекать и использовать эту информацию. Можно выявить по крайней мере три таких способа.

**Нелокальное использование информации.** Во-первых, несколько исследователей заметили, что информация, собранная в процессе поиска по дереву, обычно используется лишь локально, чтобы помочь принять решения в конкретной вершине, в которой информация была порождена. Информация о шахматной позиции, полученная динамическим анализом поддерева продолжений, обычно используется для оценивания именно этой позиции, но не для оценивания других позиций, которые могут обладать многими сходными чертами. Поэтому одни и те же факты приходится неоднократно открывать заново в различных вершинах дерева поиска. Нельзя решить этой проблемы, просто взяв информацию из контекста, в котором она возникла, и используя ее в более широком контексте, поскольку такая информация может оказаться истинной лишь для ограниченного круга ситуаций. За последние годы было предпринято несколько попыток перенести информацию из контекста, в котором она появилась, на другие подходящие контексты. Хотя пока рано судить о мощи этой идеи и о том, как именно ее следует осуществлять, она довольно многообещающа. Важное направление исследований, которое развито в работе [1], — применение анализа причин для определения множества ситуаций, в которых конкретная порция информации остается справедливой. Например, если слабость шахматной позиции может быть прослежена назад к тому ходу, в результате которого она возникла, то та же самая слабость может присутствовать и в других позициях, служащих продолжениями этого хода.

Система распознавания речи HEARSAY использует другой способ сделать информацию доступной глобально. Эта система пытается распознать фрагменты устной речи, осуществляя параллельный поиск на нескольких различных уровнях: фонетическом, лексическом, синтаксическом и семантическом. По мере того, как каждое из этих направлений поиска выдвигает и оценивает гипотезы, оно сообщает эту информацию общей «доске объявлений», которую могут читать все источники информации. Эта общая информация может быть использована, например, для отбрасывания гипотез или целого класса гипотез, которые в противном случае пришлось бы исследовать путем поиска одному из процессов. Итак, увеличение нашей способности использовать полученную поиском по дереву информацию нелокально обещает повысить интеллектуальность систем, решающих задачи.

**Семантические распознающие системы.** Вторая активно изу-

чаемая возможность повысить интеллектуальность — снабдить символьную систему обширной семантической информацией о проблемной области, с которой она имеет дело. Например, эмпирическое исследование силы игры шахматистов-мастеров показывает, что главным источником силы мастера является хранящая в его памяти информация, позволяющая ему распознать большое количество характерных особенностей позиций на доске и их комбинаций, а также информация, использующая это узнавание, чтобы предложить действия, подходящие для распознанных особенностей позиции. Эта общая идея была, разумеется, включена в шахматные программы почти с самого начала. Новым было осознание числа таких комбинаций и связанной с ними информации, которую, возможно, придется хранить для обеспечения игры на уровне мастеров: это примерно 50 000 комбинаций.

Возможность замены поиска распознаванием связана с тем, что некоторая, и особенно редкая, конфигурация может содержать огромный объем информации, если она тесно связана со структурой проблемного пространства. Когда эта структура «нерегулярна» и не допускает простого математического описания, тогда знание большого числа связанных с ней конфигураций может оказаться ключом к интеллектуальному поведению. Так это или нет для некоей конкретной проблемной области — вопрос, который проще решить эмпирическим исследованием, чем теоретически. Наш опыт работы с символьными системами, щедро наделенными семантической информацией и способностями к распознаванию образов для ее оценивания, все еще чрезвычайно ограничен.

Выше обсуждалась семантическая информация, связанная именно с системами распознавания образов. Конечно, существует также обширная область исследований по искусственному интеллекту, относящихся к обработке семантической информации и организации семантической информации посредством семантических регистров памяти; эти исследования выходят за рамки обсуждаемых здесь тем.

**Выбор подходящих представлений.** Третье направление исследований связано с возможностью сократить объем поиска или вовсе избежать его, подходящим образом выбрав проблемное пространство. Стандартный пример, ярко иллюстрирующий эту возможность, это задача об изуродованной шахматной доске. Обычная 64-клеточная шахматная доска может быть замощена в точности 32 прямоугольниками, каждый из которых закрывает ровно две клетки. Предположим теперь, что мы вырезали две клетки, находящиеся на противоположных концах большой диагонали. Можно ли эту изуродованную доску замостить в точности 31 прямоугольником? При буквально ангель-

ском терпении невозможность это сделать можно было бы доказать, перепробовав все возможные замощения. Другой способ, пригодный для менее терпеливых и более умных, состоит в том, чтобы заметить: противоположные (по диагонали) углы шахматной доски имеют одинаковый цвет. Следовательно, у изуродованной доски клеток одного цвета на две меньше, чем клеток другого цвета. Но каждый прямоугольник покрывает одну клетку одного цвета и одну клетку другого, а любой набор прямоугольников должен покрывать одинаковое число клеток каждого цвета. Поэтому решение невозможно. Как может символическая система открыть это простое индуктивное соображение вместо того, чтобы безнадежно пытаться решить задачу исчерпывающим поиском среди всех возможных замощений? Следовало бы удостоить систему, сумевшую найти это решение, высокой отметки за сообразительность.

Возможно, однако, что постановка этой задачи не освобождает нас от процессов поиска. Мы просто перенесли поиск из пространства возможных решений задачи в пространство возможных представлений. В любом случае весь процесс перехода от одного представления к другому и отыскания и оценивания представлений в значительной степени является неисследованной областью для решения задач посредством поиска. Качественные структурные принципы, управляющие представлениями, еще предстоит открыть. Поиск таких принципов почти наверняка привлечет большое внимание в предстоящее десятилетие.

## ЗАКЛЮЧЕНИЕ

Подведем итоги нашего обзора связи между символическими системами и интеллектом. Путь от «Менона» Платона к современным воззрениям был долгим, но, вероятно, можно считать обнадеживающим, что большая часть успехов на этом пути была достигнута в двадцатом веке, а значительная часть этой части — начиная с его середины. Мышление оставалось совершенно непонятным и необъяснимым до тех пор, пока современная формальная логика не предложила интерпретировать его как манипуляцию формальными символами. Казалось, что оно по-прежнему обитает в основном на небе отвлеченных платоновых идей или в столь же таинственных мирах человеческого сознания, пока компьютеры не научили нас, как машины могут обрабатывать символы. А. М. Тьюринг, памяти которого посвящено наше сегодняшнее заседание, сделал свой огромный вклад в эту проблематику как раз в середине века, на перекрестке линий развития, ведущих от формальной логики к компьютеру.



**Физические символьные системы.** Изучение логики и компьютеров открыло нам, что интеллект основан на физических символьных системах. Это наиболее фундаментальный качественный структурный принцип в информатике.

Символьные системы являются совокупностями конфигураций и процессов, причем последние могут создавать, разрушать и модифицировать первые. Наиболее важные свойства конфигураций — то, что они могут обозначать объекты, процессы или другие конфигурации, а также то, что когда они обозначают процессы, их можно интерпретировать. Интерпретация означает выполнение обозначаемых процессов. Два самых важных класса символьных систем, которые нам известны, это люди и компьютеры.

Наше современное понимание символьных систем прошло, как отмечалось выше, несколько последовательных этапов в своем развитии. Формальная логика приучила нас к символам, рассматриваемым синтаксически, как сырой материал для мышления, а также к идее манипулирования ими согласно тщательно определенным формальным процессам. Машина Тьюринга сделала синтаксическую обработку символов по-настоящему механической и подтвердила потенциальную универсальность строго определенных символьных систем. Понятие хранимой программы для компьютеров подтвердило интерпретируемость символов, уже неявно присутствующую в идее машины Тьюринга. Обработка списков выдвинула на первый план денотационные возможности символов и определила обработку символов такими способами, которые допускали независимость от фиксированной структуры физического устройства (машины). К 1956 г. все эти концепции были уже доступны, как и оборудование для их реализации. Могло начаться изучение интеллектуальности символьных систем — предмет науки об искусственном интеллекте.

**Эвристический поиск.** Второй качественный структурный принцип в искусственном интеллекте — то, что символьные системы решают задачи, порождая возможные решения и проверяя их, т.е. посредством поиска. Обычно решения отыскиваются путем создания символьных выражений и их последовательной модификации до тех пор, пока они не будут удовлетворять условиям, отвечающим решению. Итак, символьные системы решают задачи с помощью поиска. Поскольку их ресурсы ограничены, поиск не может быть выполнен мгновенно, а должен быть последовательным. Это понятие охватывает и одиночный путь от исходной точки до цели, и, если необходимы уточнения и возвраты, целое дерево таких путей.

Символьные системы не могут выглядеть интеллектуальными, когда их окружает сплошной хаос. Они проявляют интел-

лектуальность, извлекая информацию из проблемной области и используя эту информацию для направления поиска, избегая ложных поворотов и блуждания по кругу. Проблемная область должна содержать информацию, т. е. обладать некоторой упорядоченностью и структурированностью, чтобы этот метод мог работать. Парадокс из платоновского диалога «Менон» разрешается тем наблюдением, что эта информация может запоминаться, но новая информация также может извлекаться из области, обозначаемой символами. В обоих случаях первоисточник этой информации — предметная область, из которой берутся задачи.

**Эмпирические основания.** Исследования по искусственному интеллекту стремятся выяснить, как должны быть устроены символьные системы, чтобы их действия были интеллектуальными. За двадцать лет работы в этой области накопился большой объем знаний; которых хватило бы на несколько книг (и такие книги уже написаны); большей частью это весьма конкретные эмпирические данные о поведении разных типов символьных систем при решении задач из разных предметных областей. На основании этого опыта появилось, однако, и несколько обобщений, не ограниченных предметной областью или типом символьной системы, но относящихся к общим свойствам интеллекта и способам его реализации.

В настоящей лекции мы попытались сформулировать некоторые из этих обобщений. В основном они качественные, а не математические. Они больше похожи на теоретические обобщения, известные для геологии или биологии развития, чем на те, которыми оперирует теоретическая физика. Они достаточно сильны, чтобы позволить нам сегодня проектировать и строить умеренно интеллектуальные системы для широкого спектра проблемных областей, а также добиться весьма глубокого понимания того, как человеческий разум работает во многих ситуациях.

**Что дальше?** В нашем докладе мы упомянули известные нерешенные вопросы, а также поставили новые; и тех, и других довольно много. Мы не наблюдаем уменьшения энтузиазма исследований, который был характерен для этой области в течение последних 25 лет. Два ресурсных ограничения будут определять скорость прогресса в ней в течение такого же периода в будущем. Одно из них — доступные вычислительные мощности. Второе, и, вероятно, более важное, — число талантливых молодых специалистов по информатике, избирающих эту область как наиболее привлекательную и трудную из всех, им открытых.

А. М. Тьюринг закончил свою знаменитую работу «Вычислительные машины и интеллект» [5] словами:

«Мы можем заглянуть вперед лишь недалеко, но мы видим там многое, что должно быть сделано».

Многое из того, что Тьюринг в 1950 г. считал требующим осуществления, уже сделано, но в нынешней повестке дня вопросов, требующих ответа, не меньше, чем всегда. Возможно, мы вкладываем в простое утверждение, процитированное выше, более глубокий смысл, чем его автор, но нам хотелось бы думать, что в этом высказывании Тьюринг признал фундаментальную истину, которую все специалисты по информатике интуитивно знают: для всех физических символьных систем, которые, подобно всем нам, обречены на последовательный поиск в проблемной ситуации, решающий вопрос всегда один и тот же — что делать дальше?

### БЛАГОДАРНОСТЬ

Исследования авторов на протяжении многих лет поддерживались отчасти Агентством перспективных исследований Министерства обороны (руководимым Отделом научных исследований ВВС), а отчасти Национальным институтом психиатрии.

### ЛИТЕРАТУРА

1. Berliner H. Chess as problem: the development of a tactics analyzer. Ph. D. Th., Computer Sci. Dep., Carnegie-Mellon U. (1975).
2. McCarthy J. Recursive functions of symbolic expressions and their computation by machine. Comm. ACM 3, 4 (April 1960), 184—195.
3. McCulloch W. S. What is a number, that a man may know it, and a man, that he may know a number. General Semantics Bulletin, Nos. 26 and 27 (1961), 7—18.
4. Nilsson N. J. Problem Solving Methods in Artificial Intelligence. McGraw-Hill, New York, 1971.
5. Turing A. M. Computing machinery and intelligence. Mind 59 (Oct. 1950), 433—460.

### ПОСТСКРИПТУМ

РАЗМЫШЛЕНИЯ О ДЕСЯТОЙ ТЬЮРИНГОВСКОЙ ЛЕКЦИИ:  
ИНФОРМАТИКА КАК ЭМПИРИЧЕСКОЕ ИССЛЕДОВАНИЕ —  
СИМВОЛЫ И ПОИСК

Аллен Ньюэлл и Херберт А. Саймон  
Факультет информатики  
Отделение психологии  
Университет Карнеги — Меллона  
Питтсбург, Пенсильвания, 15213

Наша Тьюринговская лекция была прочитана в 1975 г., через двадцать лет после начала исследований по искусственному интеллекту в середине 50-х годов. Сейчас прошло еще десять лет. В этой лекции мы в основном

воздерживались от пророчеств и постановки задач на будущее, предпочтя вместо этого обосновать некую истину: информатика есть опытная наука. Мы сделали это, проследив историческое развитие двух общих принципов, лежащих в основе интеллектуального поведения: необходимости физических и символических систем и поиска. Можно теперь поинтересоваться, укрепило или ослабило прошедшее десятилетие позицию и оценки, выдвинутые тогда.

Многое случилось за эти десять лет. И информатика, и искусственный интеллект продолжали развиваться в научном, техническом и экономическом отношениях. Явно не высказанное, но подразумеваемое положение этой лекции состояло в том, что искусственный интеллект является частью информатики, причем обе эти дисциплины выросли на одной и той же идейной основе. В этом плане общественное развитие продолжало следовать логике (оно не всегда это делает), и искусственный интеллект продолжает оставаться частью информатики. Если и можно говорить о каких-то изменениях, то их отношения становятся все теснее по мере того, как приложения интеллектуальных систем к разработке программного обеспечения под видом экспертных систем становятся все более привлекательными.

Основное утверждение об эмпирическом исследовании отразилось во всех разделах информатики. Многие произошло повсюду, но сосредоточимся на искусственном интеллекте. Взрыв новых работ по экспертным системам, развитие исследований по обучающимся системам и разработки в области интеллектуального обучения служат важными примерами направлений, принесших крупные результаты с тех пор, как лекция была прочитана (и никак не отраженных в ней). Все они в сильнейшей степени руководствовались эмпирическими исследованиями. Даже появление работ по логическому программированию, которое в наибольшей степени связано с формальными процедурами и доказательством теорем, обязано значительной частью своей жизнеспособности превращению в программистскую деятельность — а в ней решающую роль играет практический опыт.

В развитии теории были, разумеется, значительные достижения. Особенно связанными с содержанием нашей лекции были работы по анализу сложности эвристического поиска, например недавно опубликованная книга Пирла [10]. Но это также иллюстрирует обычный заколдованный круг науки, в котором теория наконец начинает расти после того, как накопится достаточно проанализированного опыта. Нам все еще чего-то не хватает для того, чтобы замкнуть этот круг доведением теории до такого состояния, когда она обеспечивает рутинный подход к планированию дальнейших опытов, и с этого момента опытные данные и теория могут развиваться рука об руку. Это время несомненно придет, хотя пока мы до него немного не дожили.

Мы выбрали эту аудиторию, чтобы высказать два фундаментальных принципа (о символах и поиске), которые представляются нам общеизвестными в практике и понимании искусственного интеллекта, но их истинная роль еще недостаточно осознана. На самом деле это подлинный фундамент всякого интеллектуального действия. История развития этих двух принципов за последнее десятилетие была несколько различной, но оба, как нам кажется, изменялись в тех направлениях, которые подтверждают их справедливость по существу.

Выдвижение на первый план гипотезы о физических символических системах доказало свою полезность в области искусственного интеллекта, хотя мы впоследствии действительно признали разумным сформулировать эту гипотезу более подробно [7]. Эта гипотеза была принята в довольно общем смысле для выражения того взгляда на сознание, который возник в связи с появлением компьютера. Однако это не означает, что мы считаем такой подход единственно возможным. Существуют интеллектуальные позиции, отвергающие любые вычислительные подходы к сознанию и считающие упомянутую гипотезу несомненно ложной [3, 11]. Для нас важнее две другие позиции. Одна из них встречается среди философов, многие из которых верят, что

главная проблема семантики или интенциональности — как символы обозначают объекты внешнего мира — не имеет отношения к физическим символьным системам. Другая позиция встречается среди коннекционистов из специалистов по искусственному интеллекту и психологии познания, которые верят, что существуют формы организации обработки данных (реализуемые в виде нейронных сетей, перцептронов или реальных нервных систем), которые могут делать все, на что способны символьные системы, но в них невозможно обнаружить никаких сущностей, отождествимых с символами. В обоих случаях необходимы, несомненно, дополнительные исследования, и они, конечно, будут проведены. Как бы то ни было, потенциальные возможности символов по-прежнему представляются нам очень широкими, так что мы делаем ставку на гипотезу о символьных системах.

Заслуживает упоминания то развитие, которое получила гипотеза о физических символьных системах. В практике информатики и искусственного интеллекта принято описывать системы в терминах знания, которым они обладают, понимая под этим, что существуют механизмы обработки данных, вынуждающие систему вести себя так, как если бы она могла использовать это знание для достижения целей, которым система должна служить. Эта практика распространяется на проектирование, в котором задание знания, которым система должна обладать, означает задание механизмов, которые должны быть построены. Мы воспользовались возможностью; аналогичной той, которую предоставила нам Тьюринговская лекция, а именно обращение президента Американской ассоциации искусственного интеллекта (AAAI), чтобы также выразить эту практику в точных терминах [8]. Мы определили другой уровень компьютерной системы, расположенный над уровнем символов и названный нами уровнем знания. Это весьма точно соответствует тому, что в философии Дан Деннетт назвал интенциональной установкой [2]. Корни такого подхода лежат, конечно, в той замечательной особенности адаптивных систем, что их поведение полностью определяется проблемной средой, а это маскирует природу их внутренних механизмов [9, 12]. И снова мотивы, побудившие нас определить структурный уровень организации системы, который мы назвали уровнем знания, были теми же, что и в Тьюринговской лекции — выразить то, что знает каждый хороший специалист-практик в области информатики, но в такой форме, которая допускает дальнейший технический прогресс. Появились первые признаки того, что такой прогресс для уровня знания уже начинается [6].

Возвращаясь ко второй гипотезе (об эвристическом поиске), напомним, что признание важности такого поиска было явным и повсеместным в первые годы исследований по искусственному интеллекту. Нашей целью в Тьюринговской лекции было стремление подчеркнуть, что этот поиск является неотъемлемой и существенной компонентой всякого интеллектуального поведения, а не просто одним интересным механизмом среди многих других. Случилось так, что 1975 г., когда эта лекция была прочитана, как раз предшествовал расцвету той точки зрения, что знание имеет решающее значение для интеллектуального поведения. Эта тенденция нарастала с начала 70-х годов. Симптомом этого нового подхода стало возникновение новой области экспертных систем, а также новая роль инженера знаний [4]. В своих крайних проявлениях это новое движение дошло до утверждения, будто в искусственном интеллекте произошла смена парадигмы, приведшая к тому, что поиск в конце концов был отброшен, и новым ведущим принципом ИИ ныне становится знание [5].

Другая интерпретация этих перемен (та, которой придерживаемся мы) состоит в том, что никакой революции не произошло, а имел место обычный цикл аккомодации, ассимиляции и достижения равновесия, который Пиаже описывает как нормальный процесс развития понимания (хотя он говорил о развитии ребенка, а не о становлении науки). Наука развивается, расширяя каждую новую грань понимания по мере ее возникновения —

она приспособляется (аккомодация) к новому пониманию посредством напряженного стремления усвоить (ассимиляция) это понимание. Конец 70-х и начало 80-х годов были посвящены изучению того, что означает для системы обладание достаточным знанием о ее проблемной области — достаточным для того, чтобы обойтись без обширного поиска в проблемном пространстве, но все же при этом решать задачи, требующие интеллекта, а не просто реализации небольших (по объему вычислительной работы) алгоритмов. (Когда объем знаний возрастает, эти системы, конечно, начинают требовать поиска правил в базе знаний.) Соответственно задачи, решаемые этими системами, хотя они и были взяты из реального мира, отличались также небольшой интеллектуальной (т. е. требующей логического вывода) сложностью. Роль поиска в трудных интеллектуально задачах оставалась очевидной для тех, кто продолжал работать над программами с целью их завершения; этого трудно избежать, когда угроза комбинаторного взрыва маячит за каждым углом. Усвоив теперь некоторые из механизмов, позволяющих системам обладать значительными по объему базами знаний, специалисты по искусственному интеллекту, похоже, пришли к пониманию, что и поиск, и знания играют существенную роль.

Последнее наше замечание касается шахматной игры, которая неоднократно упоминалась на протяжении всей лекции, давая (как она всегда это делает) прекрасные иллюстрации многих ее положений. Достигнутый за десять лет прогресс очевиден: шахматный компьютер Hitech [1] сравнялся по силе игры с лучшими мастерами (его рейтинг 2340, тогда как у мастеров этот показатель варьирует от 2200 до 2400). Он продолжает расти, хотя никто не знает, как долго еще он будет подниматься. Система Hitech сама по себе иллюстрирует много интересных утверждений. Во-первых, она снова подтвердила роль эвристического поиска. Во-вторых, она построена на обширном поиске (200 000 позиций в секунду), так что она показывает движение именно в том направлении, которое в той лекции мы сочли ошибочным. Любопытно оказаться неправым, когда речь идет о новом научном знании. Но третий, самый важный теоретический урок, который можно извлечь из этой системы, это как раз то, что подчеркивалось в лекции, а именно: интеллектуальное поведение включает в себя сложное взаимодействие знания, полученного посредством поиска, и знания, полученного из хранимой в памяти структуры распознавания. Последние 200 очков повышения рейтинга системы Hitech — и победы, сделавшие ее знаменитой, — целиком обусловлены добавлением знания к машине с фиксированными, хотя и обширными возможностями поиска. Четвертое и последнее: поразительное мастерство системы Hitech и новые явления, которые она породила, являются еще одним свидетельством, если таковые еще требуются, что прогресс в информатике и искусственном интеллекте происходит благодаря эмпирическим исследованиям.

#### ЛИТЕРАТУРА

1. Berliner H., Ebeling C. The SUPREM architecture: A new intelligent paradigm.
2. Dennet D. C. Brainstorms. Bradford/MIT Press, Cambridge, Mass., 1978.
3. Dreyfus H. L. What Computers Can't Do: A Critique of Artificial Reason, 2nd ed. Harper and Row, New York, 1979.
4. Feigenbaum E. A. The art of artificial intelligence: Themes and case studies in knowledge engineering. In: Proceedings of the 5th International Joint Conference on Artificial Intelligence. Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa., 1977.
5. Goldstein I., Papert S. Artificial intelligence, language and the study of knowledge. Cognitive Sci., 1 (1977), 84—124.

6. Levesque H. J. Foundations of a functional approach to knowledge representation. *Artif. Intell.* 23 (1984), 155—212.
7. Newell A. Physical symbol systems. *Cognitive Sci.* 4 (1980), 135—183.
8. Newell A. The knowledge level. *Artif. Intell.* 18 (1982), 87—127.
9. Newell A., Simon H. A. Human Problem Solving. Prentice-Hall, Englewood Cliffs, N. J., 1972.
10. Pearl J. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, Reading, Mass., 1984.
11. Searle J. Minds, brains and programs. *Behav. Brain Sci.* 3 (1980), 417—457.
12. Simon H. A. The Sciences of the Artificial. MIT Press, Cambridge, Mass., 1969.

1976

## Сложность вычислений

*Микаэль О. Рабин*

Еврейский университет в Иерусалиме

Тьюринговская премия АСМ 1976 г. была вручена Микаэлю О. Рабину и Дане С. Скотту на ежегодной конференции АСМ в Хьюстоне 20 октября. Представляя лауреатов, Бернард А. Галлер, председатель комитета по премиям Тьюринга, зачитал следующее:

«Премия Тьюринга в этом году вручается Микаэлю Рабину и Дане Скотту за самостоятельный и совместный вклад, который не только наметил направление в теоретической информатике, но и явился образцом ясности и изящества для целой области исследований.

Работа Рабина и Скотта 1959 г. «Конечные автоматы и задачи их разрешения» стала классической в теории формальных языков, она представляет собой одно из самых лучших введений в эту область. Эта статья одновременно является обзором и оригинальной исследовательской работой, она технически проста и математически безупречна. Она рекомендована даже студентам младших курсов!

В последующие годы Рабин и Скотт провели самостоятельные исследования, которые подтвердили уровень их ранней работы. Применение Рабином теории автоматов к логике и разработка Скоттом непрерывной семантики для языков программирования — два примера работ, сочетающих глубину и масштабность: первая применяет информатику к математике и вторая — математику к информатике.

Рабин и Скотт показали нам, насколько хорошо математики могут помочь ученому понять свой собственный предмет. Их работа — один из великолепных образцов творческой прикладной математики».

Это была цитата из официального заключения Комитета по премиям Тьюринга, но существует менее формальная сторона сегодняшней презентации. Я хотел бы, чтобы вы поняли, что лауреаты этой премии — реальные люди, сделавшие великолепную работу, но очень похожие на всех присутствующих здесь сегодня. Профессор Микаэль Рабин родился в Германии и эмигрировал со своими родителями в Израиль в 1935 г. Он получил магистерскую степень по математике в Еврейском уни-



верситете и позднее — степень доктора философии по математике в Принстонском университете. После получения степени доктора философии он был преподавателем математики в Принстонском университете и членом исследовательского института в Принстоне. С 1958 г. он являлся преподавателем Еврейского университета в Иерусалиме. С 1972 по 1975 г. он был также ректором Еврейского университета. Ректор избирается Сенатом университета и является его научным руководителем.

Профессор Дана Скотт получил свою степень доктора философии в Принстонском университете в 1958 г. С тех пор он преподавал в Чикагском университете, Калифорнийском университете в Беркли, Станфордском университете, Амстердамском университете, Принстонском университете и Оксфордском университете в Англии.

Профессор Рабин выступит с лекцией «Вычислительная сложность», профессор Скотт — «Логика и языки программирования».

Статья Рабина помещена ниже, статья Скотта начинается на с. 65.

Очерчивается область исследований в теории сложности вычислений с акцентом на взаимосвязь между на первый взгляд различными задачами и методами. Приведены иллюстративные примеры, имеющие практическое и теоретическое значение. Обсуждаются направления новых исследований.

## 1. ВВЕДЕНИЕ

Теория сложности вычислений относится к количественным аспектам решений вычислительных задач. Обычно имеется несколько возможных алгоритмов решения таких задач, как вычисление значений алгебраических выражений, сортировка файла или синтаксический анализ цепочки символов. С каждым из этих алгоритмов связаны некоторые важные функции стоимости, такие как число шагов вычислений (как функция размера задачи), требуемый объем памяти для вычислений, размер программы и, в случае аппаратной реализации алгоритмов, — размер схемы и ее глубина.

По отношению к данной вычислительной задаче  $P$  можно ставить следующие вопросы. Какие хорошие алгоритмы существуют для решения задачи  $P$ ? Можно ли установить и доказать нижнюю оценку для какой-нибудь функции стоимости, связанной с алгоритмом? Является ли задача практически неразрешимой в том смысле, что не существует алгоритма, решающего ее в практически обозримое время? Эти вопросы можно ставить как для поведения в наихудшем случае, так и для проведения в среднем алгоритмов решения задачи  $P$ . В последнее время было предложено расширение класса рассматривае-

мых алгоритмов, включающее рандомизацию в процессе вычислений. Некоторые из упомянутых выше вопросов можно обобщить и на вероятностные алгоритмы.

Вопросы сложности интенсивно изучались в течение последних двух десятилетий как в рамках общей теории, так и для конкретных задач, важных с точки зрения математики и практики. Из многочисленных достижений упомянем следующие:

- быстрое преобразование Фурье, в последнее время сильно продвинутое, с многообразными приложениями, включая приложения к передаче информации;

- установление того, что некоторые из задач автоматического доказательства теорем, возникающих в доказательстве корректности программ, являются практически невыполнимыми;

- получение точной оценки сложности схемы сложения  $n$ -разрядных двоичных чисел;

- удивительно быстрые алгоритмы для задач комбинаторики и теории графов и их связь с синтаксическим анализом;

- серьезные сокращения времени вычислений для некоторых важных задач, полученные посредством вероятностных алгоритмов.

Несомненно, работы во всех упомянутых выше направлениях будут продолжены. Кроме того, мы предвидим ответвление от теории сложности важных новых областей. Одна из них — задача обеспечения надежно защищенной связи, нуждающаяся в новой, усиленной теории сложности, которая стала бы прочным ее фундаментом. Другая — исследование весовых функций, относящихся к структурам данных. Огромные размеры предполагаемых баз данных призывают к углублению понимания сложности, внутренне присущей таким процессам, как построение списков и поиск в них. Теория сложности формирует подходы и дает необходимые инструменты для таких исследований.

Настоящая статья, которая является расширенным вариантом Тьюринговской лекции 1976 г., имеет целью предложить читателю взгляд с высоты птичьего полета на эту жизненно важную область. Не пытаясь дать исчерпывающий обзор, мы сосредоточим внимание на основных моментах и вопросах методологии.

## 2. ТИПИЧНЫЕ ЗАДАЧИ

Мы начнем с перечисления некоторых характерных вычислительных задач, которые важны теоретически (а часто и практически) и которые были предметом интенсивного изучения и анализа. В следующих разделах мы опишем методы, приме-

нявшиеся к этим задачам, и некоторые из полученных важных результатов.

## 2.1. ЦЕЛОЧИСЛЕННЫЕ ВЫЧИСЛИМЫЕ ФУНКЦИИ НА МНОЖЕСТВЕ ЦЕЛЫХ ЧИСЕЛ

Рассмотрим функции одной или более переменных из множества  $N = \{0, 1, 2, \dots\}$  целых чисел в  $N$ . Интуитивно мы признаем, что такие функции, как  $f(x) = x!$ ,  $g(x, y) = x^2 + y^x$ , являются вычислимыми.

А. М. Тьюринг, именем которого столь уместно названы эти лекции, поставил перед собой задачу строго определить, какие именно функции  $f: N \rightarrow N$ ,  $g: N \times N \rightarrow N$  и так далее эффективно вычислимы. Его модель идеализированной вычислительной машины и класс рекурсивных функций, вычисляемых этой машиной, слишком хорошо известны, чтобы здесь на них останавливаться.

Здесь нас интересует, как измерить объем вычислительной работы, требуемой для нахождения значения  $f(n)$  вычислимой функции  $f: N \rightarrow N$ . Кроме того, возможно ли указать функции, которые сложны для вычисления любой программой? Мы вернемся к этому в п. 4.1.

## 2.2. АЛГЕБРАИЧЕСКИЕ ВЫРАЖЕНИЯ И УРАВНЕНИЯ

Пусть  $E(x_1, \dots, x_n)$  — алгебраическое выражение, построенное из переменных  $x_1, \dots, x_n$  посредством применения алгебраических операций  $+$ ,  $-$ ,  $*$ ,  $/$ . Например,  $E = (x_1 + x_2) * (x_3 + x_4)$ . Нам надо вычислить  $E(x_1, \dots, x_n)$  при  $x_1 = c_1, \dots, x_n = c_n$ . В более общем случае задача может состоять в вычислении  $k$  выражений  $E_1(x_1, \dots, x_n), \dots, E_k(x_1, \dots, x_n)$  при  $x_1 = c_1, \dots, x_n = c_n$ .

Представляют интерес следующие случаи. *Вычисление полиномов:*

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0. \quad (1)$$

*Умножение матриц*  $AB$ , где  $A$  и  $B$  суть матрицы размера  $n \times n$ . Здесь требуется найти значения  $n^2$  выражений  $a_{i1}b_{1j} + \dots + a_{in}b_{nj}$ ,  $1 \leq i, j \leq n$ , для заданных числовых значений  $a_{ij}, b_{ij}$ .

Наш пример решения уравнений есть система

$$a_{i1}x_1 + \dots + a_{in}x_n = b_i, \quad 1 \leq i \leq n, \quad (2)$$

$n$  линейных уравнений с  $n$  неизвестными  $x_1, \dots, x_n$ . Нам нужно найти (вычислить) неизвестные при заданных коэффициентах  $a_{ij}, b_{ij}$ ,  $1 \leq i, j \leq n$ .

Мы не будем здесь обсуждать интересный вопрос о приближенных решениях алгебраических и трансцендентных уравнений, к которому также применимы методы теории сложности.

### 2.3. КОМПЬЮТЕРНАЯ АРИФМЕТИКА

**Сложение.** Даны два  $n$ -разрядных числа  $a = \alpha_{n-1}\alpha_{n-2}\dots\alpha_0$ ,  $b = \beta_{n-1}\beta_{n-2}\dots\beta_0$  (например,  $n=4$ ,  $a=1011$ ,  $b=1100$ ); требуется найти  $n+1$  разрядов суммы  $a+b = \gamma_n\gamma_{n-1}\dots\gamma_0$ .

**Умножение.** Для тех же  $a$ ,  $b$  найти  $2n$  разрядов произведения  $a*b = \delta_{2n}\delta_{2n-1}\dots\delta_0$ .

Решать такие арифметические задачи можно *аппаратно*. В этом случае основание системы счисления есть 2 и  $\alpha_i, \beta_i = 0, 1$ . Для фиксированного  $n$  мы хотим построить схему с  $2n$  входами и (для сложения) с  $n+1$  выходами. Когда  $2n$  разрядов (бит) чисел  $a$  и  $b$  подаются на входы, на  $n+1$  выходах получаются  $\gamma_n, \gamma_{n-1}, \dots, \gamma_0$ . Аналогичная ситуация имеет место для умножения.

С другой стороны, мы можем рассуждать о реализации арифметики посредством алгоритма, т.е. *программным образом*. Необходимость в этом может возникать по разнообразным причинам. Например, в арифметическом устройстве может быть предусмотрено только сложение, умножение тогда должно быть реализовано подпрограммой.

Программная реализация арифметики возникает также в контексте *арифметики многократной точности*. Наш компьютер имеет длину слова  $k$ , и надо сложить и умножить числа длины  $nk$  ( $n$ -словные числа). Мы выбираем в качестве основания число  $2^k$ , так что  $0 \leq \alpha_i, \beta_i < 2^k$ , и используем алгоритмы для нахождения  $a+b$ ,  $a*b$ .

### 2.4. СИНТАКСИЧЕСКИЙ АНАЛИЗ ВЫРАЖЕНИЙ В КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКАХ

Область теории сложности ни в коей мере не ограничивается алгебраическими или арифметическими вычислениями. Рассмотрим *контекстно-свободные грамматики*, из которых в качестве примера возьмем следующую. Алфавит грамматики  $G$  состоит из символов  $t, x, y, z, (, ), +, *$ . Среди этих символов  $t$  *нетерминальный*, а все остальные — *терминальные*. *Продукции* (или правила преобразований)  $G$  суть следующие:

1.  $t \rightarrow (t+t)$ ,    2.  $t \rightarrow t*t$ ,    3.  $t \rightarrow x$ ,
4.  $t \rightarrow y$ ,        5.  $t \rightarrow z$ .

Начиная с  $t$ , мы можем последовательно преобразовывать слова, пользуясь этими продуктами. Например,

$$\bar{t} \rightarrow^1 (\bar{t}+t) \rightarrow^3 (x+\bar{t}) \rightarrow^2 (x+\bar{t}*) \rightarrow^4 (x+y*\bar{t}) \rightarrow^5 (x+y*z). \quad (3)$$

Число над стрелкой обозначает номер используемой продукции, а символом  $\bar{t}$  обозначен тот нетерминальный символ, который должен быть переписан. Последовательность вида (3) называется *выводом*, и мы говорим, что  $(x+y*z)$  выводимо из  $t$ . Множество всех слов  $u$ , выводимых из  $t$  и содержащих только терминальные символы, называется *языком, порожденным  $G$* , и обозначается  $L(G)$ . Приведенная выше  $G$  — только пример, и обобщение на произвольные контекстно-свободные грамматики очевидно.

Контекстно-свободные грамматики и языки обычно появляются в связи с языками программирования и, конечно, при анализе естественных языков. Непосредственно возникают две вычислительные задачи. По данной грамматике  $G$  и слову  $W$  (т.е. по цепочке символов) в алфавите  $G$  установить, верно ли, что  $W \in L(G)$ . Это проблема *вхождения*.

*Проблема синтаксического анализа* состоит в следующем. Для данного слова  $W$  из  $L(G)$  найти вывод (как последовательность продукций из  $G$ , подобную (3)) слова  $W$  из инициального символа  $G$ . Иначе говоря, мы хотим построить *дерево синтаксического анализа*. Нахождение дерева синтаксического анализа для алгебраического выражения, например, является существенным шагом в процессе компиляции.

## 2.5. СОРТИРОВКА ФАЙЛОВ

Файл из записей  $R_1, R_2, \dots, R_n$  хранится во внешней или основной памяти. Индекс  $i$  записи  $R_i$  указывает расположение этой записи в памяти. Каждая запись  $R$  имеет ключ (например, номер страховки в файле подоходных налогов)  $k(R)$ . Вычислительная задача состоит в переупорядочивании файла в памяти в последовательность  $R_{i_1}, R_{i_2}, \dots, R_{i_n}$  таким образом, чтобы ключи располагались в возрастающем порядке:

$$k(R_{i_1}) < k(R_{i_2}) < \dots < k(R_{i_n}).$$

Мы подчеркиваем как различие между ключом и записью, которая может быть существенно длиннее ключа, так и требование действительного переупорядочивания записей. Эти особенности делают проблему более реалистичной и более сложной, чем просто сортировка чисел.

## 2.6. МАШИННОЕ ДОКАЗАТЕЛЬСТВО ТЕОРЕМ

С возникновением компьютеров появилось вполне понятное желание наделить их способностью рассуждать, что привело к приложению значительных усилий в этом направлении. В ча-

стности, были предприняты попытки сделать компьютер способным к логическим выводам и математическим рассуждениям и осуществить это посредством доказательства теорем чистой логики или посредством вывода теорем в математических теориях. Мы рассмотрим важный пример теории сложения натуральных чисел.

Рассмотрим систему  $\mathcal{N} = \langle N, + \rangle$ , состоящую из натуральных чисел  $N = \{0, 1, \dots\}$  и операции сложения  $+$ . Формальный язык  $L$ , употребляемый для обсуждения свойств  $\mathcal{N}$ , — это так называемый язык исчисления предикатов первого порядка. Он имеет переменные  $x, y, z, \dots$ , значениями которых являются натуральные числа, символ операции  $+$ , равенство  $=$ , обычные пропозициональные связки и кванторы  $\forall$  («для всех» и  $\exists$  («существует»)).

Такое предложение, как  $\exists x \forall y (x + y = y)$ , есть формальная запись высказывания: «Существует число  $x$ , такое, что для всех чисел  $y$   $x + y = y$ ». Это предложение фактически истинно в  $\mathcal{N}$ .

Множество всех предложений языка  $L$ , истинных в  $\mathcal{N}$ , будем называть теорией  $\mathcal{N}$  ( $Th(\mathcal{N})$ ) и обозначать  $PA = Th(\mathcal{N})$ . Например,

$$\forall x \forall y \exists z [x + z = y \vee y + z = x] \in PA.$$

Мы будем также пользоваться названием «арифметика Пресбургера», введенным в честь Пресбургера, получившего важные результаты для  $Th(\mathcal{N})$ .

*Проблема разрешения* для  $PA$  состоит в нахождении алгоритма, если в самом деле такой алгоритм существует, определяющего для каждого заданного предложения  $F$  языка  $L$ , верно  $F \in PA$  или нет.

Такой алгоритм для  $PA$  построил Пресбургер [12]. Со времени появления этой работы многие исследователи пытались построить эффективные алгоритмы для этой задачи и реализовать их программно. Эти усилия предпринимались часто в рамках конкретных проектов в области автоматизации программирования и верификации программ. Это делалось потому, что свойства программ, которые пытаются установить, иногда сводимы к утверждениям о сложении натуральных чисел.

### 3. ЦЕНТРАЛЬНЫЕ ВОПРОСЫ И МЕТОДОЛОГИЯ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ

В предыдущем разделе мы перечислили некоторые типичные вычислительные задачи. Позднее мы представим результаты, которые были получены для них. Здесь мы опишем в общем виде главные вопросы, которые там возникают, и центральные понятия теории сложности.

### 3.1. ОСНОВНЫЕ ПОНЯТИЯ

Класс однородных вычислительных задач мы будем называть *проблемой*. Индивидуальные случаи проблемы  $P$  мы будем называть *частными случаями* проблемы  $P$ . Таким образом,  $P$  есть множество всех своих частных случаев. Такое описание проблемы есть только предмет соглашения и удобства обозначений. Мы можем, например, говорить о проблеме умножения матриц. Частные случаи этой проблемы суть (для любого целого  $n$ ) пары матриц, которые нужно перемножить.

С каждым частным случаем  $I \in P$  проблемы  $P$  мы связываем размер  $|I|$  (обычно целое число). Эта функция  $|I|$  не единственна, и ее выбор диктуется теоретическими и практическими соображениями, связанными с тем, с какой точки зрения интересна эта проблема.

Возвращаясь к примеру умножения матриц, отметим, что разумной мерой для пары  $I = (A, B)$  ( $n \times n$ )-матриц, которые надо перемножить, является  $|I| = n$ . Если мы изучаем объем памяти, требующейся для алгоритма умножения матриц, то подходящей может быть мера  $|I| = n^2$ . Наоборот, не кажется правдоподобным, что функция размера  $|I| = n^n$  может естественным образом возникнуть в каком-нибудь контексте.

Пусть  $P$  — проблема и  $AL$  — алгоритм, решающий ее. При решении частного случая  $I \in P$  алгоритм  $AL$  выполняет некоторую последовательность вычислений  $S_I$ . С  $S_I$  мы связываем некоторые числовые характеристики. Существенными являются, например, следующие характеристики: (1) длина  $S_I$ , которая характеризует время вычисления; (2) глубина  $S_I$ , т.е. число уровней параллельных шагов, на которые  $S_I$  может быть разложена; она соответствует времени, которое  $S_I$  потребовалось бы при параллельных вычислениях; (3) объем памяти, требуемый для вычисления  $S_I$ ; (4) вместо общего числа шагов в  $S_I$  мы можем подсчитывать число шагов некоторого вида, таких, как арифметические операции при алгебраических вычислениях, число сравнений при сортировке или число обращений к памяти.

Для аппаратной реализации алгоритмов мы обычно определяем размер  $|I|$ , так, чтобы все частные случаи  $I$  одинакового размера  $n$  решались при помощи одной и той же схемы  $C_n$ . Сложность схемы  $C$  определяется разными способами, например, как число элементов, глубина, снова связанная с временем вычислений, или выбираются другие меры сложности, такие как число модулей, связанное с технологией, используемой при построении схемы.

После того как выбрана мера  $\mu$  вычисления  $S$ , функция  $F_{AL}$  сложности вычисления может быть определена несколькими

ми способами, два главных из них — сложность в *наихудшем случае* и сложность *поведения в среднем*. Первое понятие определяется следующим образом:

$$F_{AL}(n) = \max \{ \mu(S_I) \mid I \in P, |I| = n \}. \quad (4)$$

Для того чтобы определить поведение в среднем, мы задаем распределение вероятностей  $p$  на каждом множестве  $P_n = \{I \mid I \in P, |I| = n\}$ . Так, для  $I \in P, |I| = n$ , величины  $p(I)$  — вероятность появления  $I$  среди всех других частных случаев размера  $n$ . Поведение в среднем алгоритма  $AL$  тогда определяется так:

$$M_{AL}(n) = \sum_{I \in P_n} p(I) \mu(S_I). \quad (5)$$

В п. 4.7 мы рассмотрим применимость предположения о распределении вероятностей.

*Анализ* алгоритмов связан со следующим вопросом. Для заданной функции размера  $|I|$  и меры вычисления  $\mu(S_I)$  точно определить для данного алгоритма  $AL$ , решающего проблему  $P$ , либо сложность  $F_{AL}$  для *наихудшего случая*, либо, при подходящих предположениях, поведение в среднем  $M_{AL}$ . В настоящей статье мы не будем рассматривать вопросы анализа, а предположим, что функция сложности известна или по крайней мере достаточно хорошо определена для наших целей.

### 3.2. ВОПРОСЫ

Теперь мы располагаем всеми необходимыми понятиями для постановки основного вопроса теории сложности: насколько успешно или с какой стоимостью может быть решена заданная вычислительная проблема  $P$ ? Мы не имеем в виду никакого конкретного алгоритма решения  $P$ . Наша цель — рассмотреть *все* возможные алгоритмы решения  $P$  и попытаться сформулировать утверждение о вычислительной сложности, внутренне присущей  $P$ . Не надо забывать, что предварительный шаг в изучении сложности проблемы  $P$  — выбор меры  $\mu(S)$ , которая должна быть использована. Иначе говоря, мы должны решить, исходя из математических или практических соображений, *какую именно сложность* мы хотим исследовать. Сделав такой выбор, мы идем дальше.

В общих чертах (более детальные примеры и иллюстрации пока отложим) опишем основные интересующие нас вопросы. Кроме последнего пункта, они окажутся сгруппированными в пары.

(1) Найти эффективные алгоритмы для проблемы  $P$ .



(2) Установить нижние оценки сложности, внутренне присущей  $P$ .

(3) Поискать точные решения  $P$ .

(4) Найти алгоритмы для приближенных решений.

(5) Изучить сложность для наихудшего случая.

(6) Изучить сложность  $P$  в среднем.

(7) Найти последовательные алгоритмы для  $P$ .

(8) Найти параллельные алгоритмы для  $P$ .

(9) Изучить программно реализованные алгоритмы.

(10) Изучить аппаратно реализованные алгоритмы.

(11) Рассмотреть возможность решения посредством вероятностных алгоритмов.

Под (1) мы понимаем поиск хороших с точки зрения практики алгоритмов для данной проблемы. Дело в том, что непосредственно очевидные алгоритмы часто можно заменить алгоритмами гораздо лучшими. Улучшения в 100 раз вполне возможны. Но даже экономия вдвое может иногда означать различие между осуществимостью и неосуществимостью.

В то время как всякий алгоритм  $AL$  для  $P$  дает *верхнюю оценку* величины  $F_{AL}$  сложности  $P$ , нас интересует *нижняя оценка*. Типичный результат состоит в том, что всякий алгоритм  $AL$ , решающий  $P$ , удовлетворяет неравенству  $g(n) \leq F_{AL}(n)$  по крайней мере для  $n_0 < n$ , где  $n_0 = n_0(AL)$ . В некоторых удачных случаях верхняя и нижняя оценки совпадают. Сложность такой проблемы в этом случае известна полностью. В любой ситуации знание нижней оценки представляет интерес математический и, кроме того, руководит нами в поиске хороших алгоритмов, указывая, какие попытки заведомо будут безуспешны.

Идея (4) приближенного решения проблемы существенна по той причине, что иногда практически удовлетворительное приближенное решение гораздо проще вычислить, чем точное решение.

Основные вопросы (1) и (2) можно изучать в комбинации с одной или несколькими вопросами (3) — (11). Так, например, мы можем изучать верхнюю оценку среднего времени, требуемого для сортировки  $n$  процессорами, работающими параллельно. Или мы можем изучать число логических элементов, необходимых для сортировки  $n$  входных двоичных разрядов.

Может показаться, что с многообразными возможностями выбора меры сложности и разнообразными вопросами, которые могут возникать, теория сложности вычислений станет коллекцией изолированных результатов и не связанных друг с другом методов. Положение, которое мы пытаемся подчеркнуть приводимыми ниже примерами, — это большая степень взаимосвязи

разных вопросов внутри этой области и общность идей и методов, здесь преобладающих.

Мы увидим, что эффективные алгоритмы для параллельных вычислений полиномов переносятся на схемы для быстрого сложения  $n$ -разрядных двоичных чисел. Идея быстрого преобразования Фурье приводит к хорошим алгоритмам для умножения чисел с многократной точностью. На более высоком уровне отношение между программными и аппаратными алгоритмами вполне аналогично отношению между последовательными и параллельными вычислениями. Современные программы построены в расчете на единственный процессор и потому являются последовательными, тогда как аппаратная реализация содержит много идентичных подсистем, которые можно рассматривать как примитивные процессоры, работающие параллельно. В наших примерах снова и снова возникает метод предварительной обработки, еще раз показывая общность всех возникающих в данной области вопросов.

## 4. РЕЗУЛЬТАТЫ

### 4.1. СЛОЖНОСТЬ ОБЩЕРЕКУРСИВНЫХ ФУНКЦИЙ

В [13, 14] автором начато изучение классификации вычислимых целочисленных функций по сложности их вычисления. Подход был принят аксиоматический, так что понятия и результаты применимы ко всякому осмысленному классу алгоритмов и всякой мере сложности, определенной на множестве вычислений.

Пусть  $K$  — класс алгоритмов, возможно базирующийся на некоторой модели математических машин, так что для каждой вычислимой функции  $f: N \rightarrow N$  существует алгоритм  $AL \in K$ , вычисляющий ее. Мы не уточняем меру  $\mu(S)$  на вычислениях  $S$ , а предполагаем, что  $\mu$  удовлетворяет некоторым естественным аксиомам. Эти аксиомы выполняются на всех конкретных примерах мер в п. 3.1. В качестве размера целого числа  $n$  берется  $|n| = n$ . Вычисление функции  $f$  есть проблема, где для каждого частного случая  $n$  мы должны найти  $f(n)$ . По (4) из п. 3.1 мы имеем для каждого алгоритма  $AL$  для  $f$  функцию  $F_{AL}(n)$  сложности вычисления, измеряющую работу, требуемую для вычисления  $f$  по алгоритму  $AL$ .

**Теорема** [13, 14]. Для всякой вычислимой функции  $g: M \rightarrow N$  существует вычислимая функция  $f: N \rightarrow \{0, 1\}$ , такая, что для всякого алгоритма  $AL \in K$ , вычисляющего  $f$ , существует число  $n_0$ , такое, что

$$g(n) < F_{AL}(n) \text{ для } n_0 < n. \quad (6)$$

Мы требуем, чтобы  $f$  была функцией со значениями 0 и 1, потому что в противном случае мы могли бы построить сложную функцию, просто позволив  $f(n)$  расти очень быстро, так что записывать результат станет слишком сложно.

Ограничение  $n_0 < n$  в (6) необходимо. Для каждого  $f$  и  $k$  мы можем построить алгоритм, содержащий внутри себя таблицу значений  $f(n)$ ,  $n \leq k$ , делающий вычисления тривиальными для  $n \leq k$ .

Основной смысл приведенной теоремы заключается в том, что (6) при подходящем  $n_0 = n_0(AL)$  имеет место для *любого* алгоритма вычисления  $f$ . Таким образом, внутренне присущая сложность вычисления  $f$  больше, чем  $g$ .

Начиная с [14], Блум [1] ввел другие, но по существу эквивалентные аксиомы для функции сложности. Блум получил много интересных результатов, включая теорему об *ускорении*. Эта теорема показывает существование вычислимых функций, для которых не существует наилучшего алгоритма. Точнее, для всякого алгоритма вычисления такой функции найдется другой алгоритм, вычисляющий ее гораздо быстрее.

Исследования в этой области абстрактной теории сложности значительно продвинулись в последнее десятилетие. Это послужило основой для теории сложности вычислений по двум причинам: во-первых, из-за постановки самого вопроса о стоимости вычислений и, во-вторых, подчеркиванием необходимости решать и сравнивать все возможные алгоритмы решения данной проблемы.

С другой стороны, абстрактная теория сложности не ведет к пониманию конкретных вычислительных задач и их оценке с точки зрения практики. Это показывают следующие примеры.

#### 4.2. АЛГЕБРАИЧЕСКИЕ ВЫЧИСЛЕНИЯ

Начнем с примера вычисления полиномов. Выберем в качестве меры число арифметических операций и обозначим через  $(nA, kM)$  стоимость  $n$  сложений/вычитаний и  $k$  умножений/делений. Переписывая полином (1) в виде

$$f(x) = (\dots((a_n x + a_{n-1})x + a_{n-2}))x + \dots + a_0,$$

мы убеждаемся, что общий полином  $n$ -й степени можно вычислить за  $(nA, kM)$ . В духе вопросов (1) и (2) п. 3.2 мы спрашиваем, может ли искусный алгоритм использовать меньшее число операций. Довольно тонкие математические рассуждения показывают, что приведенное выше число операций оптимально, так что вопрос полностью решен.

Т. Моцкин в [9] ввел в рассмотрение важную идею *предварительной обработки* для вычислений. Во многих важных прило-

жениях требуется вычислить один и тот же полином при многих различных значениях аргументов  $x=c_1, x=c_2, \dots$ . Он предложил следующую стратегию предварительной обработки коэффициентов полинома (1). Вычислить раз и навсегда все числа  $\alpha_0(a_0, \dots, a_n), \dots, \alpha_n(a_0, \dots, a_n)$  по данным коэффициентам  $a_0, \dots, a_n$ . При вычислении  $f(c)$  использовать  $\alpha_0, \dots, \alpha_n$ . Такой подход имеет смысл в том случае, когда стоимость предварительной обработки мала по сравнению с общей экономией при вычислениях  $f(c_1), f(c_2), \dots$ , т. е. когда ожидаемое число значений аргументов, для которых  $f(x)$  должно быть вычислено, очень велико. Моцкин получил следующий результат.

**Теорема.** *С использованием предварительной обработки полином степени  $n$  можно вычислить за  $(nA, (\lfloor n/2 \rfloor + 2)M)$ .*

Снова можно доказать, что этот результат по существу наилучший из возможных. Что можно сказать о параллельных вычислениях? Если мы используем  $k$  процессоров и должны вычислить выражение, требующее по крайней мере  $m$  операций, то самое лучшее, на что можно надеяться, — это время вычислений  $(m/k) - 1 + \log_2 k + O(1)$ .

А именно, предположим, что все процессоры непрерывно заняты, тогда  $m - k$  операций выполняются за время  $(m/k) - 1$ . Оставшиеся  $k$  операций должны объединяться посредством бинарных операций над  $k$  входами в один выход, и это требует времени по крайней мере  $\log_2 k$ . Ввиду вышесказанного следующий результат, принадлежащий Манро и Патерсону [10], близок к наилучшему.

**Теорема.** *Полином (1) может быть вычислен  $k$  процессорами, работающими параллельно, за время  $(2n/k) + \log_2 k + O(1)$ .*

Прогресс в развитии аппаратных средств позволяет надеяться, что мы сможем употреблять большое число процессоров для одной задачи. Брент [3] наряду с другими изучал следствия неограниченного параллелизма и доказал следующее.

**Теорема.** *Пусть  $E(x_1, \dots, x_n)$  — арифметическое выражение, где каждая переменная появляется лишь однажды. Выражение  $E$  может быть вычислено при условии неограниченного параллелизма за время  $4 \log_2 n$ .*

Другой важный объект — быстрое преобразование Фурье (БПФ). Операция свертки, которая имеет много приложений, таких, как преобразование сигналов, — пример вычисления, сильно упрощаемого применением БПФ. Пусть  $a_1, \dots, a_n$  — последовательность  $n$  чисел,  $b_1, b_2, \dots$  — входной поток чисел. Определим для  $i = 1, 2, \dots$

$$c_i = a_1 b_i + a_2 b_{i+1} + \dots + a_n b_{i+n-1}. \quad (7)$$

Нам нужно вычислить значения  $c_1, c_2, \dots$ . Из (7) может показаться, что стоимость вычисления одного  $c_i$  равна  $2n$  операций. Если мы вычисляем величины  $c_i$  блоками размера  $n$ , т.е.  $c_1, \dots, c_n$ , затем  $c_{n+1}, \dots, c_{2n}$  и т.д., используя БПФ, то стоимость вычисления каждого блока примерно равна  $5n \log_2 n$ , так что стоимость одного  $c_i$  равна  $5 \log_2 n$ .

Используя хитроумную комбинацию алгебраических и теоретико-числовых идей, Виноград [20] недавно улучшил время вычисления свертки для малых  $n$  и дискретного преобразования Фурье для малых и средних  $n$ . Для  $n \sim 1000$ , например, его метод примерно вдвое быстрее традиционного алгоритма БПФ.

Очевидные методы для умножения  $(n \times n)$ -матриц и для решения системы. (2)  $n$  линейных уравнений с  $n$  неизвестными требует примерно  $n^3$  операций. Штрассен [17] получил следующий удивительный результат.

**Теорема.** *Две  $(n \times n)$ -матрицы могут быть перемножены с использованием самое большое  $4,7n^{2,81}$  операций. Система  $n$  линейных уравнений с  $n$  неизвестными может быть решена посредством  $4,8n^{2,81}$  операций.*

Маловероятно, что показатель  $\log_2 7 \sim 2,81$  на самом деле наилучший из возможных, но ко времени написания данной статьи все попытки улучшить этот результат не дали успеха.

### 4.3. НАСКОЛЬКО БЫСТРО МЫ МОЖЕМ СКЛАДЫВАТЬ И УМНОЖАТЬ?

Этот очевидно важный вопрос подвергся тщательному анализу. Простое рассуждение показывает, что если использовать элементы с  $r$  входами, то схема сложения  $n$ -разрядных двоичных чисел требует временной сложности по крайней мере  $\log_r n$ . Эта нижняя оценка фактически достижима.

Следует отметить, что в ходе замечаний п. 3.2 об аналогии между параллельными алгоритмами и аппаратной реализацией алгоритмов один из наилучших результатов по схемам сложения (Брент [2]) использует булевы тождества, которые немедленно переводимы в эффективный алгоритм параллельного вычисления полиномов.

Приведенные выше результаты относятся к двоичному представлению чисел, которые нужно сложить. Может ли быть, что при достаточно разумном кодировании чисел  $0 \leq a < 2^n$  сложение по модулю  $2^n$  выполнимо быстрее, чем за  $\log_r n$ ? На этот вопрос ответил Виноград [19]. При весьма общих предположениях относительно кодирования нижняя оценка остается  $\log_r n$ .

Если обратиться к арифметике многократной точности, то интересные вопросы возникают в связи с умножением. Очевид-

ный метод перемножения чисел длины  $n$  содержит  $n^2$  двоичных операций. Ранние попытки улучшения использовали простые алгебраические тождества и привели к упрощению до  $O(n^{1,58})$  операций.

Шенхаге и Штрассен [16] использовали связь между умножением натуральных чисел и умножением многочленов и применили быстрое преобразование Фурье (БПФ) для получения следующей теоремы.

**Теорема.** *Два  $n$ -разрядных двоичных числа могут быть перемножены за  $O(n \log n \log \log n)$  операций.*

Получение нижних оценок сложности умножения целых чисел должно относиться к конкретной вычислительной модели. При очень разумных предположениях Патерсон, Фишер и Мейер [11] серьезно уменьшили разрыв между верхней и нижней оценками, показав следующее.

**Теорема.** *Для умножения  $n$ -разрядных двоичных чисел необходимо по крайней мере  $O(n \log n / \log \log n)$  операций.*

#### 4.4. СКОРОСТЬ СИНТАКСИЧЕСКОГО АНАЛИЗА

Синтаксический анализ выражений в контекстно-свободных грамматиках на первый взгляд кажется требующим дорогостоящих возвратных вычислений. Динамическое вычисление, которое одновременно отслеживает предков всех подцепочек анализируемой цепочки, приводит к алгоритму, требующему  $O(n^3)$  шагов для синтаксического анализа слова длины  $n$ . Коэффициент при  $n^3$  зависит от грамматики. Долгое время этот результат оставался наилучшим, хотя для специальных классов контекстно-свободных грамматик были получены и лучшие оценки.

Фишер и Мейер заметили, что на основе штрассеновского алгоритма матричного умножения можно получить алгоритм поразрядного умножения двух булевых  $(n \times n)$ -матриц, требующий  $O(n^{2,81}c(n))$  поразрядных булевых операций. Здесь  $c(n) = \log n \log \log n \log \log \log n$  и, таким образом, имеет вид  $O(n^a)$  для всякого  $0 < a$ .

Валиант [18] обнаружил, что синтаксический анализ столь же сложен, как и умножение булевых матриц. Следовательно, поскольку  $\log_2 7 < 2,81$ , справедлива следующая теорема.

**Теорема.** *Выражения длины  $n$  в контекстно-свободном языке  $L(G)$  могут быть разобраны за время  $d(G)n^{2,81}$ .*

Мы снова видим, как результаты из алгебраической теории сложности приносят плоды в области комбинаторных вычислений.

#### 4.5. ОБРАБОТКА ДАННЫХ

Из области приложений теории сложности к обработке данных мы рассмотрим известнейший пример, а именно сортировку. Мы следуем формулировкам, данным в п. 2.5.

Хорошо известно, что сортировка  $n$  чисел в памяти с произвольным доступом требует примерно  $n \log n$  сравнений. Это и худший случай поведения некоторых алгоритмов, и поведение в среднем других алгоритмов в предположении, что все перестановки равновероятны.

Переупорядочивание записей  $R_1, R_2, \dots, R_n$  ставит дополнительные проблемы, потому что файл обычно расположен в последовательной или близкой к последовательной памяти, такой, как магнитная лента или диск. Ограниченность оперативной памяти позволяет переносить в нее для переупорядочивания лишь небольшое число записей за один раз. Все же можно разработать алгоритмы физического переупорядочивания файлов за время  $cn \log n$ , где  $c$  зависит от характеристик рассматриваемой системы.

Один поучительный результат в этой области принадлежит Флойду [6]. В его модели файл располагается на нескольких страницах  $P_1, \dots, P_m$  и каждая страница содержит  $k$  записей, так что  $P_i$  содержит записи  $R_{i1}, \dots, R_{ik}$ . Для наших целей мы можем предположить без ограничения общности, что  $m=k$ . Задача состоит в перераспределении записей таким образом, что  $R_{ij}$  окажется на странице  $P_j$  для всех  $1 \leq i, j \leq k$ . Объем быстрой памяти позволяет прочитать две страницы  $P_i, P_j$ , переупорядочить их записи и вывести обе страницы на внешний носитель. Используя рекурсию, аналогичную применяемой в БПФ, Флойд доказал следующее.

**Теорема** *Переупорядочивание записей описанным выше способом может быть выполнено за  $k \log_2 k$  пересылок в быструю память. Этот результат наилучший из возможных.*

Нижняя оценка устанавливается посредством рассмотрения подходящей энтропийной функции. Она применима в предположении, что внутри быстрой памяти записи только перетасовываются. Неизвестно, можно ли построить алгоритм с меньшим числом пересылок страниц, если допустить вычисления с записями, рассматриваемыми как цепочки бит.

#### 4.6. ПРАКТИЧЕСКИ НЕВЫПОЛНИМЫЕ ЗАДАЧИ

Машинное доказательство теорем порождает вычислительные задачи, которые требуют столь большого числа вычислительных шагов, что являются практически невыполнимыми.

В попытках прогонять программы для решения проблемы разрешения пресбургеровской арифметики (РА) на ЭВМ вычисления заканчивались лишь в простейших случаях. Теоретическое обоснование этого эмпирического факта дает следующий результат Фишера и Рабина [5].

**Теорема.** *Существует константа  $c > 0$ , такая, что для каждого разрешающего алгоритма AL для РА существует число  $n_0$ , такое, что для каждого  $n > n_0$  существует предложение  $H$  языка  $L$  (языка для сложения чисел), удовлетворяющее следующим условиям: (1)  $l(H) = n$ , (2) AL требует более чем  $2^{2^{cn}}$  шагов для определения того, верно ли  $H \in RA$ , т.е. верно ли, что  $H$  истинно в  $\langle N, + \rangle$ . Здесь  $l(H)$  обозначает длину  $H$ .*

Константа  $c$  зависит от обозначений, используемых при формулировании свойств  $\langle N, + \rangle$ . Во всяком случае, она не слишком мала. Очень быстрый рост неустранимой нижней оценки  $2^{2^{cn}}$  показывает, что, даже пытаясь решить проблему разрешения для этой очень простой и «элементарной» математической теории, мы сталкиваемся с практически невыполнимыми вычислениями. Мейер [8] привел примеры теорий с еще более ошеломляюще сложными проблемами разрешения.

Простейший уровень логического вывода — исчисление высказываний. Из пропозициональных переменных  $p_1, p_2, \dots$  мы можем строить формулы, такие, как  $[p_1 \wedge \sim p_1] \vee [p_2 \wedge \sim p_2]$ , посредством пропозициональных связок. Проблема выполнимости состоит в выяснении для пропозициональной формулы  $G(p_1, \dots, p_n)$  существования таких истинностных значений переменных  $p_1, \dots, p_n$ , что формула  $G$  истинна. Например, значения  $p_1 = F$  (ложь),  $p_n = T$  (истина) удовлетворяют приведенной выше формуле.

Прямой алгоритм для проблемы выполнимости требует примерно  $2^n$  шагов для формулы с  $n$  переменными. Неизвестно, существуют ли неэкспоненциальные алгоритмы для проблемы выполнимости.

Большая важность этого вопроса была осознана в результате исследований Кука [4]. Можно определить естественный процесс так называемого полиномиального сведения одной вычислительной задачи  $P$  к другой задаче  $Q$ . Если  $P$  полиномиально сводима к  $Q$  и  $Q$  разрешима за полиномиальное время, тогда то же справедливо относительно  $P$ . Две взаимно сводимые задачи называются полиномиально эквивалентными. Кук показал, что проблема выполнимости эквивалентна так называемой задаче клик в графах. Карп [7] указал большое число задач, эквивалентных проблеме выполнимости. Среди них такие задачи, как двоичное целочисленное программирование, существо-



вание гамильтонова цикла в графе, целочисленная задача о коммивояжере и многие другие.

В силу таких эквивалентностей если за полиномиальное время разрешима одна из этих задач, то все остальные тоже разрешимы. Вопрос о том, имеет ли выполнимость полиномиальную сложность, называется проблемой  $P=NP$  и справедливо считается самой знаменитой задачей теории сложности вычислений.

#### 4.7. ВЕРОЯТНОСТНЫЕ АЛГОРИТМЫ

Как упомянуто в п. 3.1, изучение поведения в среднем, т.е. ожидаемого времени работы алгоритма, основано на предположении о распределении вероятностей в пространстве частных случаев задачи. Это предположение наталкивается на некоторые методологические трудности. Мы можем постулировать некоторое распределение, например считать все частные случаи равновероятными, но на практике источник частных случаев решаемой задачи может вести себя совершенно иначе. Распределение может меняться во времени и часто будет нам неизвестным. В экстремальной ситуации большинство частных случаев, которые действительно встречаются, суть в точности те, для которых алгоритм работает хуже всего.

Можно ли воспользоваться вероятностными моделями вычислений каким-нибудь другим способом, при котором мы будем полностью контролировать ситуацию? *Вероятностный алгоритм*  $AL$  для задачи  $P$  использует датчик случайных чисел. Когда решается частный случай  $I \in P$ , генерируется короткая последовательность  $r = (b_1, \dots, b_k)$  случайных чисел, и они используются в  $AL$  для *точного* решения  $P$ . После фиксации случайного выбора  $r$  алгоритм выполняется вполне детерминированно.

Мы говорим, что такой  $AL$  решает  $P$  за ожидаемое время  $f(n)$ , если для всякого  $I \in P$ ,  $|I| = n$ ,  $AL$  решает  $I$  за ожидаемое время, меньшее или равное  $f(n)$ . Под ожидаемым временем мы понимаем среднее время решения  $I$  посредством  $AL$  для всех возможных способов выбора последовательности  $r$  (которые мы предполагаем равновероятными).

Отметим различие между этим понятием и хорошо известным методом Монте-Карло. В последнем методе мы строим для задачи стохастический процесс, который ее моделирует, и, «измеряя» его, получаем приближенные решения задачи. Поэтому метод Монте-Карло есть по существу аналоговый метод решения. Наши вероятностные алгоритмы, наоборот, используют случайные числа  $b_1, \dots, b_k$ , чтобы определить ветвление в алгоритмах, в остальном детерминированных, и получить точное, а не приближенное решение.

Может показаться маловероятным, что такая консультация с «бросанием кости» может ускорить вычисления. Автор систематически изучал [15] вероятностные алгоритмы. Оказывается, что в некоторых случаях этот подход дает драматические улучшения.

Ближайшей парой в множестве точек  $x_1, \dots, x_n \in R^k$  ( $k$ -мерного пространства) называется пара  $x_i, x_j, i \neq j$ , для которой расстояние  $d(x_i, x_j)$  минимально. Вероятностный алгоритм находит ближайшую пару за ожидаемое время  $O(n)$ , т.е. быстрее любого обычного алгоритма.

Задача определения того, является ли натуральное число  $n$  простым, становится нерешаемой для больших  $n$ . Известные методы не срабатывают при  $n \sim 10^{60}$ , если они применяются не к числам специального вида. Вероятностный алгоритм, построенный автором, работает со скоростью  $O((\log n)^3)$ . На компьютере средней мощности число  $2^{400} - 593$  было распознано как простое за несколько минут. Этот метод работает столь же хорошо для многих других чисел сравнимого размера.

Полный потенциал этих идей еще не известен и достоин дальнейшего изучения.

## 5. НОВЫЕ НАПРАВЛЕНИЯ

Из возможных путей дальнейших исследований упомянем только два.

### 5.1. БОЛЬШИЕ СТРУКТУРЫ ДАННЫХ

Коммерческие нужды требуют создания еще больших баз данных. В то же время нынешние и, более того, возникающие технологии будущего позволяют создавать гигантскую память с разными степенями свободы доступа.

Многие текущие исследования по базам данных нацелены на языки взаимодействия между пользователем и системой. Но огромные размеры списков и других предполагаемых структур могут сделать требуемые операции над этими структурами очень дорогостоящими, если не будет достигнуто углубленное понимание алгоритмов выполнения этих операций.

Мы можем начать с проблемы нахождения теоретической, но в то же время практически значимой модели таких списков. Эта модель должна быть достаточно гибкой, чтобы ее можно было применять к различным типам используемых сейчас структур списков.

Какие операции применяются к спискам? Мы можем перечислить некоторые: поиск в списке, сбор мусора, доступ к различным точкам списка, вставки, исключения, слияние списков.

Можно ли систематизировать изучение этих и других важных операций? Каковы разумные функции стоимости, связанные с этими операциями?

Наконец, глубокое количественное понимание структур данных могло бы быть основой рекомендаций для выбора архитектуры компьютеров. Дают ли параллельные вычисления заметное ускорение выполнения различных операций над структурами данных? Какими полезными свойствами можно наделить списки при использовании ассоциативной памяти? Это, разумеется, только примеры.

## 5.2. ЗАЩИЩЕННЫЕ ЛИНИИ СВЯЗИ

В защищенных линиях связи применяются некоторые схемы кодирования, и мы можем поставить основные вопросы теории сложности вычислений по отношению к таким системам. Проиллюстрируем это посредством системы блочного кодирования.

В блочном кодировании употребляют цифровые устройства, которые входные слова длины  $n$  кодируют при помощи ключа. Если  $x$  — слово длины  $n$  и  $z$  — ключ (предположим, что ключи тоже длины  $n$ ), то пусть  $E_z = y$ ,  $l(y) = n$ , обозначает результат кодирования  $x$  ключом  $z$ . Сообщение  $w = x_1, x_2, \dots, x_k$  длины  $kn$  кодируется как  $E_z(x_1)E_z(x_2) \dots E_z(x_k)$ .

Если противник способен добыть текущий ключ  $z$ , он может декодировать связи между сторонами, так как мы предполагаем, что он владеет кодирующим и декодирующим оборудованием. Он может также вставить поддельные сообщения, которые будут правильно закодированы. В коммерческой связи это даже более опасно, чем нарушение секретности.

С точки зрения секретности следует также принимать в расчет возможность того, что противник владеет некоторым количеством сообщений  $w_1, w_2, \dots$  открытым текстом и в закодированном виде. Может ли ключ  $z$  быть вычислен по этим данным?

Недостаточно доказать, что такое вычисление невыполнимо. Дело в том, что результаты современной теории сложности дают нам информацию для наихудшего случая. Так, если, скажем, для большинства вычислений, связанных с поисками ключа, установлена нижняя оценка вычислительной сложности  $2^n$ , то задача считается невыполнимой. Но если алгоритм обнаружит ключ в практически обозримое время в одном случае из тысячи, то возможности обмана будут недопустимо велики.

Таким образом, нам необходима такая теория сложности, которая позволяет нам установить и доказать, что некоторое вычисление неосуществимо практически всегда. Например, система блочного кодирования надежна, если любой алгоритм

определения ключа будет заканчиваться в практически обозримое время только в  $O(2^{-n})$  случаев. Мы весьма далеки от создания такой теории, особенно на данном этапе, когда проблема  $P=NP$  еще не решена.

#### ЛИТЕРАТУРА

1. Blum M. A machine independent theory of the complexity of recursive functions. J. ACM 14 [1967], 322—336.
2. Brent R. P. On the addition of binary numbers. IEEE Trans. Computrs. C-19 [1970], 758—759.
3. Brent R. P. The parallel evaluation of algebraic expressions in logarithmic time. Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, Ed., Academic Press, New York, 1973, pp. 83—102.
4. Cook S. A. The complexity of theorem proving procedures. Proc. Third Annual ACM Symp. on Theory of Comptng. 1971, pp. 151—158.
5. Fisher M. J., and Rabin M. O. Super-exponential complexity of Presburger arithmetic. In Complexity of Computations (SIAM—AMS Proc., Vol. 7), R. M. Karp, Ed., 1974, pp. 27—41.
6. Floyd R. W. Permuting information in idealized two-level storage. In Complexity of Computer Computations. R. Miller and J. Thatcher, Eds., Plenum Press, New York, 1972, pp. 105—109.
7. Karp R. M. Reductibility among combinatorial problems. In Complexity of Computer Computations. R. Miller and J. Thatcher, Eds., Plenum Press, New York, 1972, pp. 85—103.
8. Meyer A. R. The inherent computational complexity of theories of order. Proc. Int. Cong. Math., Vol. 2, Vancouver, 1974, pp. 477—482.
9. Motzkin T. S. Evaluation of polynomials and evaluation of rational functions. Bull. Amer. Math. Soc. 61 (1955), 163.
10. Munro I., and Paterson, M. Optimal algorithms for parallel polynomial evaluation. J. Computr. Syst. Sci. 7 (1973), 189—198.
11. Paterson M., Fisher, M. J., and Meyer, A. R. An improved overlap argument for on-line multiplication. Proj. MAC Tech. Report 40, M. I. T. (1974).
12. Presburger M. Ueber die Vollstaendigkeit eines gewissen Systems Arithmetik ganzer Zahlen in welchem die Addition als einzige Operation hervortritt. Comptes-rendues du I Congres de Mathematiciens de Pays Slaves, Warsaw, 1930, pp. 92—101, 395.
13. Rabin M. O. Speed of computation and classification of recursive sets. Third Convention Sci. Soc., Israel, 1959, pp. 1—2.
14. Rabin M. O. Degree of difficulty of computing a function and a partial ordering of recursive sets. Tech. Rep. No. 1, O. N. R., Jerusalem, 1960.
15. Rabin M. O. Probabilistic algorithms. In Algorithms and Complexity, New Directions and Recent Trends, J. F. Traub, Ed., Academic Press, New York, 1976, pp. 29—39.
16. Schonhage A., and Strassen, V. Schnelle Multiplication grosser Zahlen. Computing 7 (1971), 281—292.
17. Strassen V. Gaussian elimination is not optimal. Num. Math. 13 (1969), 354—356.
18. Valiant L. G. General context-free recognition in less than cubic time. Rep., Dept. Computr. Sci., Carnegie-Mellon U., Pittsburgh, Pa., 1974.
19. Winograd S. On the time required to perform addition. J. ACM 12 (1965), 277—285.
20. Winograd S. On computing the discrete Fourier transform. Proc. Natl. Acad. Sci. USA 73 (1976), 1005—1006.

1979

## Нотация как средство мышления

*К. Е. Айверсон*

Исследовательский центр Томаса Уотсона фирмы IBM

Тьюринговская премия АСМ за 1979 г. была вручена К. Е. Айверсону председателем Тьюринговского комитета В. Карлсоном на ежегодной конференции АСМ в Детройте, шт. Мичиган, 29 октября 1979 г.

Этим выбором Комитет по премиям за общие технические достижения отметил пионерские работы Айверсона по языкам программирования и математической нотации, завершившиеся созданием языка, известного теперь под названием APL. Был отмечен также вклад Айверсона в реализацию интерактивных систем, в обучение средствами языка APL и в теорию и практику языков программирования.

Айверсон родился и вырос в Канаде. Докторскую степень он получил в 1954 г. в Гарвардском университете. Там он работал младшим профессором прикладной математики с 1955 по 1960 г. Затем он заключил контракт с фирмой IBM и в 1970 г. был введен в научный совет IBM в знак признания его вклада в развитие языка APL.

В настоящее время доктор Айверсон работает в компании И. П. Шарпа в Торонто. Он опубликовал множество статей по языкам программирования и написал четыре книги о программировании и математике: «Язык программирования» (1962 г.), «Элементарные функции» (1966 г.), «Алгебра: алгоритмическая интерпретация» (1972 г.) и «Элементарный анализ» (1976 г.).

\* \* \*

Уже давно признана важность классификации, нотации и языка как орудий мышления. Например, в химии и ботанике системы классификации Лавуазье и Линнея во многом стимулировали дальнейшие исследования и определили направления. Относительно языка Дж. Буль в [1, стр. 24] утверждает: «Всеми признана истина, что язык является инструментом человеческого мышления, а не только средством выражения готовых мыслей».

Математическая нотация представляет собой, может быть, самый известный и самый развитый пример языка, специально служащего орудием мышления. Признание важной роли нотации в математике явственно следует из высказываний математиков, приводимых в [2, стр. 332, 331]. Эти высказывания за-

служивают того, чтобы прочесть их полностью, но об их общем духе можно судить по следующим цитатам:

«Освобождая мозг от всей необязательной работы, хорошая нотация позволяет ему сосредоточиться на более сложных проблемах и в результате увеличивает умственную мощь цивилизации».

А. Н. Уайтхейд

«Количество смысла, спрессованного в малом пространстве алгебраическими знаками, является еще одним обстоятельством, облегчающим рассуждения, которые мы привыкли выполнять с их помощью».

Чарльз Бэббидж

Тем не менее математическая нотация страдает существенными недостатками. В частности, ей недостает универсальности, и она должна интерпретироваться по-разному — в зависимости от предмета, автора и даже непосредственного контекста. Поскольку языки программирования предназначены для управления компьютерами, они обладают важными преимуществами как средства мышления. Они не только универсальны (имеют общее назначение), но к тому же исполнимы и недвусмысленны. Их исполнимость позволяет использовать компьютеры для пространственных экспериментов над идеями, выраженными на языке программирования, а отсутствие двусмысленностей делает возможными точные мысленные эксперименты. Однако в других отношениях большинство языков программирования, несомненно, слабее математической нотации и мало применяются в качестве инструментов мышления способами, которые, скажем, математик-прикладник счел бы важными.

Основная идея данной работы состоит в том, что обнаруживаемые в языках программирования исполнимость и универсальность могут быть эффективно объединены в одном согласованном языке с преимуществами, предоставляемыми математической нотацией.

(а) В разд. 1 идентифицируются наиболее существенные характеристики математической нотации и на простых примерах иллюстрируются способы представления таких характеристик в исполнимой нотации.

(б) В разд. 2 и 3 это иллюстрирование продолжается с более глубоким рассмотрением ряда аспектов, выбранных из-за общей привлекательности и полезности. В разд. 2 речь идет о полиномах, а в разд. 3 разбираются преобразования представлений, касающиеся ряда тем, в том числе перестановок и ориентированных графов. Хотя эти темы можно назвать математическими, они непосредственно связаны с программированием для компьютеров, а эта связь возрастает по мере того, как продолжается развитие программирования в законную математическую дисциплину.

(в) В разд. 4 приводятся примеры тождеств и формальных доказательств. Многие из этих формальных доказательств относятся к тождествам, которые были неформально установлены и использовались в предыдущих разделах.

(г) В заключительном разделе представлены некоторые общие сопоставления с математической нотацией, упоминаются подходы к другим темам и обсуждается проблема введения обозначений в контекст какой-то конкретной предметной области.

В качестве исполнимого языка мы будем применять APL, язык общего назначения, который был создан с целью обеспечения ясного и точного выражения мыслей при написании и преподавании и который был реализован в виде языка программирования только после нескольких лет его использования и развития [3].

Несмотря на то что среди читателей многие незнакомы с языком APL, я предпочел не давать здесь отдельное введение в APL, а вводить его в контекст по мере надобности. Математическая нотация всегда вводится таким образом, а не излагается в отдельном курсе подобно языкам программирования. Нотация, подходящая в качестве инструмента общения в какой-либо области, должна легко вводиться в контекст этой предметной области. Одно из преимуществ введения языка APL в наш контекст состоит в том, что читатель получает возможность оценить трудность такого введения.

Однако введение в контекст не сравнимо с полным обсуждением всех оттенков каждого элемента нотации, и читатель должен быть готов либо обобщать эти описания очевидным и систематическим способом в соответствии с требованиями дальнейших приложений, либо обратиться к литературе. Вся используемая здесь нотация суммирована в приложении А и полностью представлена на с. 24—60 руководства [4].

Читатели, имеющие доступ к компьютеру, оснащенный системой APL, могут захотеть перевести функциональные описания, приводимые здесь в форме прямого описания [5, с. 10] («с использованием знаков  $\alpha$  и  $\omega$  для представления левых и правых аргументов»), в каноническую форму, которая требуется для исполнения. Функция для автоматического выполнения этого преобразования приведена в приложении Б.

## 1. ВАЖНЫЕ ХАРАКТЕРИСТИКИ НОТАЦИИ

Помимо выделенных во введении исполнимости и универсальности хорошая нотация должна обладать характеристиками, знакомыми каждому пользователю математической нотации:

- Легкость выражения конструкций, возникающих в данном классе задач.
- Содержательность.
- Возможность управлять выражением конкретных деталей.
- Экономичность.
- Пригодность для формальных доказательств.

Я не утверждаю, что этот список исчерпывающий, но буду вести последующее обсуждение в его рамках.

Недвусмысленная исполнимость введенной нотации сохраняет важность и будет подчеркиваться приведением под выражениями порождаемых ими результатов в явном виде. Чтобы различать выражения и результаты, мы будем задавать выражения в том виде, как они автоматически представляются на компьютерах с языком APL. Например, *целая функция*  $\iota$ , примененная к аргументу  $N$ , порождает вектор первых  $N$  целых чисел; *сведение к сумме*, обозначаемое знаками  $+ /$ , порождает сумму элементов своего векторного аргумента, и мы будем пользоваться следующим изображением:

$$\begin{array}{cccccc} & & & & 15 & \\ & & & & & \\ 1 & 2 & 3 & 4 & 5 & \\ & & & & + / 15 & \\ 15 & & & & & \end{array}$$

Мы будем применять один неисполнимый элемент нотации: символ  $\leftrightarrow$  между двумя выражениями устанавливает их эквивалентность.

### 1.1. ЛЕГКОСТЬ ВЫРАЖЕНИЯ КОНСТРУКЦИИ, ВОЗНИКАЮЩИХ В ЗАДАЧАХ

Чтобы быть эффективным средством мышления, нотация должна допускать удобное выражение не только тех понятий, которые непосредственно вытекают из постановки задачи, но и тех, которые возникают при последующем анализе, обобщении и специализации.

Например, рассмотрим показанную на рис. 1 кристаллическую структуру, в которой атомы из последовательных слоев лежат не непосредственно один на другом, а образуют «плотную упаковку» с атомами из соответствующего нижнего слоя. Поэтому количество атомов в последовательных слоях, начиная с вершины рис. 1, задается как  $\iota 5$ , а общее количество задается как  $+ / \iota 5$ .

Трехмерная структура такого кристалла также дает плотную упаковку. Атомы в плоскости, лежащей поверх рис. 1, были бы расположены между атомами нижней плоскости; базовая ячейка состояла бы из четырех атомов. Иначе говоря, полная



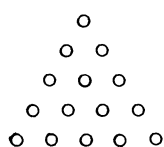


Рис. 1.

трехмерная структура, соответствующая рис. 1, представляла бы собой набор тетраэдров, плоскости которых имеют длины оснований 1, 2, 3, 4 и 5. Поэтому числа элементов в последовательных плоскостях равны *частичным* суммам вектора  $\iota 5$ , т.е. первого элемента, сумме первых двух элементов и т.д. Такие частичные суммы вектора  $V$  обозначаются как  $+\backslash V$ , причем функция  $+\backslash$  называется *разверткой суммы*.

$$\begin{array}{r}
 +\backslash \iota 5 \\
 1 \ 3 \ 6 \ 10 \ 15 \\
 +/+ \backslash \iota 5 \\
 35
 \end{array}$$

Последнее выражение дает общее число атомов в тетраэдре.

Сумма  $+/ \iota 5$  может быть представлена графически и другими способами, например как показано на рис. 2 слева. В сочетании с такой же, но перевернутой конфигурацией справа эта сумма может просто интерпретироваться как число единиц в прямоугольнике, т.е. как произведение.

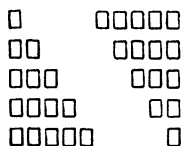


Рис. 2.

Длины строк фигуры, образованной слиянием двух частей рис. 2, задаются сложением вектора  $\iota 5$  с обращенным вариантом того же вектора. Итак:

$$\begin{array}{r}
 \iota 5 \\
 1 \ 2 \ 3 \ 4 \ 5 \\
 \phi \iota 5 \\
 5 \ 4 \ 3 \ 2 \ 1 \\
 (\iota 5) + (\phi \iota 5) \\
 6 \ 6 \ 6 \ 6 \ 6
 \end{array}$$

Эта конфигурация, состоящая из пяти повторений числа 6, может быть выражена как  $5\rho 6$ , и мы имеем

$$\begin{array}{r}
 5\rho 6 \\
 6 \ 6 \ 6 \ 6 \ 6 \\
 +/5\rho 6 \\
 30 \\
 6 \times 5 \\
 30
 \end{array}$$

Факт  $+ / 5 \rho 6 \leftrightarrow 6 \times 5$  следует из определения умножения как повторяемого сложения.

Из предыдущего следует, что  $+ / 1 5 \leftrightarrow (6 \times 5) \div 2$  и, в более общем виде, что

$$+ / 1 N \leftrightarrow ((N + 1) \times N) \div 2. \quad A.1$$

## 1.2. СОДЕРЖАТЕЛЬНОСТЬ

Нотация называется *содержательной*, если формы выражений, возникающих в некоем множестве задач, подсказывают соответствующие выражения, находящие применение в других задачах. Теперь мы рассмотрим другие, аналогичные приведенным применения введенных до сих пор функций, а именно:

$$1 \quad \phi \quad \rho \quad + / \quad + \backslash$$

Пример

$$\begin{array}{ccccccc} & & & & 5 \rho 2 & & \\ & & & & 2 & 2 & 2 \\ & & & & \times / 5 \rho 2 & & \\ & & & & 3 2 & & \end{array}$$

наводит на мысль, что  $\times / M \rho N \leftrightarrow N * M$ , где знак  $*$  представляет степенную функцию. Итак, сходство определений степени в терминах умножения и умножения в терминах сложения может быть показано следующим образом:

$$\begin{array}{l} \times / M \rho N \leftarrow \rightarrow N * M \\ + / M \rho N \leftarrow \rightarrow N \times M \end{array}$$

Аналогичные выражения для частичных сумм и частичных произведений можно разработать так:

$$\begin{array}{ccccccc} & & & & \times \backslash 5 \rho 2 & & \\ & & & & 2 & 4 & 8 \\ & & & & 16 & 32 & \\ & & & & 2 * 1 5 & & \\ & & & & 2 & 4 & 8 \\ & & & & 16 & 32 & \end{array}$$

$$\begin{array}{l} \times \backslash M \rho N \leftrightarrow N * 1 M \\ + \backslash M \rho N \leftrightarrow N \times 1 M \end{array}$$

Поскольку они представимы треугольником, изображенным на рис. 1, суммы  $+ \backslash 1 5$  называются *треугольными* числами. Это частный случай *фигурных* чисел, получаемых повторяемым применением развертки суммы, начиная либо с  $+ \backslash 1 N$ , либо с  $+ \backslash N \rho 1$ . Итак:

$$\begin{array}{rcccl}
 & & 5\rho 1 & & +\backslash\backslash 5\rho 1 \\
 1 & 1 & 1 & 1 & 1 \\
 & & & & 1 \quad 3 \quad 6 \quad 10 \quad 15 \\
 & & +\backslash 5\rho 1 & & +\backslash\backslash\backslash 5\rho 1 \\
 1 & 2 & 3 & 4 & 5 \\
 & & & & 1 \quad 4 \quad 10 \quad 20 \quad 35
 \end{array}$$

Замена сумм для последовательных целых на произведения порождает факториалы, например:

$$\begin{array}{rcccl}
 & & 15 & & \\
 1 & 2 & 3 & 4 & 5 \\
 & & \times / 15 & & \times \backslash 15 \\
 120 & & & & 1 \quad 2 \quad 6 \quad 24 \quad 120 \\
 & & ! 5 & & ! 15 \\
 120 & & & & 1 \quad 2 \quad 6 \quad 24 \quad 120
 \end{array}$$

Выразительная сила языка отчасти основывается на возможности представления тождеств в кратких, общих и легко запоминаемых формах. Мы проиллюстрируем это утверждение, выразив *двойственность* в форме, включающей законы Де-Моргана, умножение — применив логарифмы, а также приведя другие, менее известные тождества.

Если  $V$  — вектор положительных чисел, то произведение  $\times/V$  можно получить, взяв натуральные логарифмы от каждого элемента из  $V$  (обозначаемые как  $\oplus V$ ), просуммировав их  $(+/\oplus V)$  и применив экспоненциальную функцию  $(*\backslash/\oplus V)$ . Итак:

$$\times/V \leftrightarrow *\backslash/\oplus V$$

Поскольку экспоненциальная функция  $*$  — это функция, обратная к натуральному логарифму  $\oplus$ , то правая часть тождества предлагает общую форму

$$IG \ F/G \ V$$

где  $IG$  — функция, обратная  $G$ .

Используя знаки  $\wedge$  и  $\vee$  для обозначения функций *и* и *или*, а знак  $\sim$  для обозначения обратной к самой себе функции логического отрицания, мы можем выразить законы Де-Моргана для произвольного числа элементов так:

$$\begin{array}{l}
 \wedge / B \leftarrow \rightarrow \sim \vee / \sim B \\
 \vee / B \leftarrow \rightarrow \sim \wedge / \sim B
 \end{array}$$

Конечно, элементы из  $B$  ограничиваются логическими значениями 0 и 1. Воспользовавшись символами отношения для обозначения *функций* (например,  $X < Y$  порождает 1, если  $X$  меньше

чем  $Y$ , и  $0$  в противном случае), мы можем выразить другие двойственности, такие, как

$$\neq / B \leftarrow \rightarrow \sim = / \sim B$$

$$= / B \leftarrow \rightarrow \sim | \neq / \sim B$$

Наконец, используя  $\sqcap$  и  $\sqcup$  для обозначения функций *максимума* и *минимума*, мы можем выразить двойственности с включением арифметического отрицания:

$$\sqcap / V \leftarrow \rightarrow \neg \sqcup / \neg V$$

$$\sqcup / V \leftarrow \rightarrow \sqcap / \neg V$$

Можно отметить также, что в любой из приведенных выше двойственностей развертка ( $F \setminus$ ) может заменить сведение ( $F /$ ).

### 1.3. ДЕТАЛИЗАЦИЯ

Как заметил Бэббидж в высказывании, цитируемом в [2], краткость облегчает рассуждения. Краткость достигается детализацией, и мы рассмотрим здесь три важных способа ее обеспечения: использование массивов, присваивание имен функциям и переменным, употребление операций.

Мы уже видели примеры краткости, обеспечиваемой одномерными массивами (векторами), при рассмотрении двойственности; дальнейшая детализация достигается путем использования матриц и других массивов более высокого ранга, поскольку функции, определенные на векторах, систематически обобщаются на массивы более высокого ранга.

В частности, можно специфицировать ось, к которой применяется функция. Например,  $\Phi[1]M$  действует по первой оси матрицы  $M$  и обращает каждый из столбцов,  $\Phi[2]$  обращает каждую строку;  $M, [1]N$  сочленяет столбцы [помещая  $M$  над  $N$ ], а  $M, [2]N$  сочленяет строки;  $a + / [1]M$  суммирует столбцы, а  $a + / [2]M$  суммирует строки. Если никакая ось не специфицирована, функция применяется вдоль последней оси. Так,  $+ / M$  суммирует строки. И наконец, сведение вдоль *первой оси* может обозначаться символом  $\dagger$ .

Можно различать два способа использования имен: *постоянные* имена, соотносимые с фиксированными объектами, применяются для объектов, помещенных в весьма общих ситуациях, а *целевые* имена присваиваются (посредством символа  $\leftarrow$ ) величинам, представляющим интерес в более узком контексте. Например, постоянная (имя) 144 соотносится с фиксированным объектом, а имена CRATE, LAYER и ROW, присваемые выражениями

$$\text{CRATE} \leftarrow 144$$

$$\text{LAYER} \leftarrow \text{CRATE} \div 8$$

$$\text{ROW} \leftarrow \text{LAYER} \div 3$$

являются целевыми именами, или именами *переменных*. Предусматриваются также постоянные имена для векторов, например 2 3 5 7 11 для числового вектора из пяти элементов и «ABCDE» для символьного вектора из пяти элементов.

Аналогичные различия существуют между именами функций. Такие постоянные имена, как +,  $\times$  и \*, присваиваются так называемым *примитивным* функциям общего назначения. Детализированные описания, например  $+/M_p N$  для  $N \times M$  и  $\times/M_p N$  для  $N * M$ , подчиняются постоянным именам  $\times$  и \*.

Менее известные примеры постоянных имен функций связываются с запятой, которая *сцепляет* свои аргументы, как в примере

$$(\iota 5), (\Phi 5) \leftarrow \rightarrow 1\ 2\ 3\ 4\ 5\ 5\ 4\ 3\ 2\ 1$$

и с функцией *базового представления*  $\top$ , которая порождает представление своего правого аргумента по основанию, заданному ее левым аргументом.

Например:

$$2\ 2\ 2\ \top\ 3 \leftrightarrow 0\ 1\ 1$$

$$2\ 2\ 2\ \top\ 4 \leftrightarrow 1\ 0\ 0$$

$$BN \leftarrow 2\ 2\ 2\ \top\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7$$

$$BN$$

$$0\ 0\ 0\ 0\ 1\ 1\ 1\ 1$$

$$0\ 0\ 1\ 1\ 0\ 0\ 1\ 1$$

$$0\ 1\ 0\ 1\ 0\ 1\ 0\ 1$$

$$BN, \Phi BN$$

$$0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0$$

$$0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0$$

$$0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0$$

Матрица  $BN$  имеет важное значение, потому что ее можно рассматривать несколькими способами. Помимо представления двоичных чисел столбцы представляют все подмножества множества из трех элементов, а также входы в таблицы истинности для трех логических аргументов. Легко увидеть, что общее выражение для  $N$  имеет вид  $(N_p 2) \top (\iota 2 * N) = 1$ , и мы можем захотеть присвоить этой функции целевое имя. Посредством непосредственного описания (приложение Б) этой функции при-

сваивается имя  $\underline{T}$  следующим образом:

$$\underline{T} : (\omega \rho 2) \top (\iota 2 * \omega) \rightarrow 1. \quad \text{A.2}$$

Символ  $\omega$  представляет аргумент функции; в случае двух аргументов левый представляется как  $\alpha$ . В соответствии с таким описанием функции  $\underline{T}$  выражение  $\underline{T}3$  порождает показанную выше булеву матрицу  $BN$ .

Три выражения, разделенные двоеточиями, служат также для следующего описания функции: выражение из середины выполняется первым; если его значение есть нуль, то выполняется первое выражение, в противном случае выполняется последнее выражение. Эта форма удобна для рекурсивных описаний, в которых функция используется для своего собственного описания. Например, функция, порождающая биномиальные коэффициенты для порядка, который указывается в ее аргументе, может быть определена рекурсивно так:

$$BC : (X, 0) + (0, X \leftarrow BC \ \omega - 1) : \omega = 0 : 1. \quad \text{A.3}$$

Итак,  $BC \ 0 \leftarrow \rightarrow 1$ ,  $BC \ 1 \leftarrow \rightarrow 1 \ 1$ , а  $BC \ 4 \leftarrow \rightarrow 1 \ 4 \ 6 \ 4 \ 1$ .

Термин *операция*, используемый в строгом математическом смысле, а не как примерный синоним *функции*, относится к объекту, который применяется к функциям для получения функций; примером является операция дифференцирования.

Мы уже встречались с операциями *сведения* и *развертки*, обозначенными как  $/$  и  $\backslash$ , и видели, как они способствуют краткости при их применении к различным функциям для получения семейств связанных функций, например  $+ /$ ,  $\times /$  и  $\wedge /$ . Теперь мы продолжим пояснение этого понятия, введя операцию *внутреннего произведения*, обозначаемую точкой. Функция (например,  $+ /$ ), порожденная неким оператором, будет называться *выведенной функцией*.

Если  $P$  и  $Q$  — два вектора, то внутреннее произведение  $+ \cdot \times$  описывается так:

$$P + \cdot \times Q \leftarrow \rightarrow + / P \times Q$$

и аналогичные описания имеют место и для других пар функций, отличных от  $+$  и  $\times$ . Например:

$$\begin{array}{rcl} & P + 2 & 3 \ 5 \\ & Q + 2 & 1 \ 2 \\ & P + \cdot \times Q & \\ 17 & & \\ & P \times \cdot * Q & \\ 300 & & \\ & P \downarrow \cdot + Q & \\ 4 & & \end{array}$$

Каждое из приведенных выше выражений имеет по крайней мере одну полезную интерпретацию:  $P \dot{+} \times Q$  представляет собой общую стоимость порядковых величин  $Q$  для объектов, цены которых заданы через  $P$ ; если  $P$  — это вектор простых чисел, то  $P \times \cdot Q$  является числом, разложение которого на простые множители задается степенями  $Q$ ; если  $P$  задает расстояние от исходного места до пунктов пересадки, а  $Q$  задает расстояния от пунктов пересадки до места назначения, то  $P \underline{\cdot} \dot{+} Q$  дает минимальное возможное расстояние.

Функция  $\dot{+} \times$  эквивалентна внутреннему произведению или математическому скалярному произведению и обобщается на матрицы, как в математике. Аналогично обобщаются и другие варианты, например  $\times \cdot$ . Так, если  $\underline{T}$  — это функция, описанная формулой A.2, то

$$\begin{array}{rcc} & \underline{T} & 3 \\ \begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \\ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \end{array} & & \\ \begin{array}{r} P \dot{+} \cdot \underline{T} & 3 \\ 0 \ 5 \ 3 \ 8 \ 2 \ 7 \ 5 \ 10 \end{array} & & \begin{array}{r} P \times \cdot \underline{T} & 3 \\ 1 \ 5 \ 3 \ 15 \ 2 \ 10 \ 6 \ 30 \end{array} \end{array}$$

Эти примеры выявляют важное обстоятельство: если значения  $B$  булевы, то  $P \dot{+} \times B$  порождает суммы над подмножествами из  $P$ , специфицируемыми значениями 1 из  $B$ , а  $P \times \cdot B$  порождает произведения над подмножествами.

Вариант  $\circ \times$  представляет собой специальное применение операции внутреннего произведения для получения выведенной функции, которая дает произведения каждого элемента своего левого аргумента и каждого элемента правого аргумента. Так:

$$\begin{array}{rcc} & 2 & 3 & 5 \circ \times 15 \\ \begin{array}{r} 2 \ 4 \ 6 \ 8 \ 10 \\ 3 \ 6 \ 9 \ 12 \ 15 \\ 5 \ 10 \ 15 \ 20 \ 25 \end{array} & & \end{array}$$

Функция  $\circ \times$  называется *внешним произведением*, как и в тензорном анализе, а такие функции, как  $\circ \dot{+}$  и  $\circ \cdot$ , описываются аналогичным образом и порождают «таблицы функций» для соответствующих функций.

Например:

$$\begin{array}{rcc} D \dot{+} 0 & 1 & 2 & 3 \\ \begin{array}{r} D \circ \cdot \lceil D \\ 0 \ 1 \ 2 \ 3 \\ 1 \ 1 \ 2 \ 3 \\ 2 \ 2 \ 2 \ 3 \\ 3 \ 3 \ 3 \ 3 \end{array} & & \begin{array}{r} D \circ \cdot \geq D \\ 1 \ 0 \ 0 \ 0 \\ 1 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 0 \\ 1 \ 1 \ 1 \ 1 \end{array} & & \begin{array}{r} D \circ \cdot ! D \\ 1 \ 1 \ 1 \ 1 \\ 0 \ 1 \ 2 \ 3 \\ 0 \ 0 \ 1 \ 3 \\ 0 \ 0 \ 0 \ 1 \end{array} \end{array}$$

Символ ! обозначает функцию биномиальных коэффициентов, а таблица  $D \circ ! D$  содержит треугольник Паскаля с вершиной слева; при обобщении на отрицательные аргументы (как в случае  $D \leftarrow -3 -2 -1 0 1 2 3$ ) она будет содержать треугольные числа, а также фигурные числа для фигур более высокого порядка. Такое обобщение на отрицательные аргументы представляет интерес и для других функций. Например, таблица  $D \circ \times D$  состоит из четырех квадрантов, разделенных строкой и столбцом из нулей, причем эти квадранты наглядно демонстрируют правило знаков для умножения.

Структура этих таблиц функций проявляет другие свойства этих функций, позволяющие кратко формировать утверждения, доказываемые перебором. Например, коммуникативность проявляется как симметрия относительно диагонали. Более точно, если результат применения к таблице  $T \leftarrow D \circ f D$  функции транспонирования  $\mathbb{Q}$  (которая обращает порядок осей своего аргумента) совпадает с  $T$ , то функция  $f$  коммуникативна на данной области. Так,  $T = \mathbb{Q} T \leftarrow D \circ \Gamma D$  порождает таблицу из единиц, потому что функция  $\Gamma$  коммутативная.

Соответствующие тесты ассоциативности требуют таблицы третьего ранга вида  $D \circ f(D \circ f D)$  и  $(D \circ f D) \circ f D$ . Например,

$D \leftarrow 0 \ 1$							
$D \circ \wedge (D \circ \wedge D)$		$(D \circ \wedge D) \circ \wedge D$		$D \circ \leq (D \circ \leq D)$		$(D \circ \leq D) \circ \leq D$	
0 0	0 0	0 0	0 0	1 1	1 1	0 1	0 1
0 0	0 0	0 0	0 0	1 1	1 1	0 1	0 1
0 0	0 0	0 0	0 0	1 1	1 1	1 1	1 1
0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1

#### 1.4. ЭКОНОМИЧНОСТЬ

Полезность языка в качестве средства мышления возрастает с расширением круга проблем, к которым он подходит, но она уменьшается с ростом словарного запаса и сложности грамматических правил, которые пользователь должен хранить в своей памяти. Поэтому важное значение имеет экономичность нотации.

Экономичность требует, чтобы большое количество идей выражалось в терминах относительно малого словаря. Фундаментальная схема достижения этого состоит во введении грамматических правил, с помощью которых содержательные фразы и предложения могут конструироваться посредством комбинирования элементов словаря.

Эта схема может быть проиллюстрирована на первом рассмотренном примере — относительно простое и весьма полезное



понятие суммы первых  $N$  целых чисел было введено не как примитив, а как фраза, сконструированная из двух общих понятий, функции  $!$  порождения вектора целых и функции  $+/$  суммирования элементов вектора. Более того, выведенная функция  $+/$  сама является фразой, причем суммирование — это выведенная функция, сконструированная из более общего понятия оператора сведения, примененного к конкретной функции.

Экономичность достигается также за счет общности введенных функций. Например, описание функции факториала, обозначаемой через  $!$ , не ограничивается целыми числами, и поэтому гамма-функция от  $X$  может быть записана как  $!X-1$ . Аналогично отношения, определенные для всех вещественных аргументов, обеспечивают при их применении к логическим аргументам несколько важных логических функций: исключающее или ( $\neq$ ), материальная импликация ( $\leq$ ) и эквивалентность ( $=$ ).

Достижимую для рассмотренных до сих пор понятий экономичность можно подтвердить, вспомнив введенный словарь:

$!$	$\rho$	$\Phi$	$\tau$	$,$
$/$	$\backslash$	$.$		
$+ - \times \div * \bullet ! \lceil \lfloor \Phi$ $\vee \wedge \sim < \leq = > \neq$				

Первостепенный интерес представляют пять функций и три операции, перечисленные в первых двух строках, остальные известные функции были введены, чтобы проиллюстрировать многосторонность операций.

В отличие от экономии функций значительная экономия символов достигается за счет того, что любой символ может представлять как *одноместную функцию* (т. е. функцию от одного аргумента), так и *двуместную функцию* аналогично тому, как знак минус обычно используется и для вычитания, и для отрицания. Поскольку две представленные функции могут оказаться связанными, как в случае знака минус, облегчается запоминание символов.

Например,  $X*Y$  и  $*Y$  представляют степень и экспоненту,  $X\otimes Y$  и  $\otimes Y$  представляют логарифм по основанию  $X$  и натуральный логарифм,  $X\div Y$  и  $\div Y$  представляют деление и получение обратного числа, а  $X!Y$  и  $!Y$  представляют функцию биномиальных коэффициентов и факториал (т. е.  $X!Y \leftarrow \rightarrow (!Y) \div (!X) \times X(!Y-X)$ ). Символ  $\rho$ , используемый для двуместной функции повторения, представляет также и одноместную функцию, которая дает форму аргумента (т. е.  $X \leftarrow \rightarrow \rho X \rho Y$ ), символ  $\Phi$ , служащий для одноместной функции обращения, представляет также двуместную функцию циклической перестановки, демонстрируе-

мую с помощью  $2\phi 15 \leftarrow \rightarrow 34512$  и  $-2\phi 15 \leftarrow \rightarrow 45123$  и наконец, запятая представляет не только конкатенацию, но и одноместное *связывание*, порождающее вектор элементов своего аргумента в порядке «старшинства строк». Например:

$$\begin{array}{ccc} T & 2 & , T & 2 \\ 0011 & & 00110101 \\ 0101 & & \end{array}$$

Важное значение имеет также простота грамматических правил нотации. Поскольку использованные до сих пор правила были знакомы нам по математической нотации, они не выделялись явно, но следует отметить два упрощения порядка исполнения:

- (1) Все функции интерпретируются единообразно, и отсутствуют правила старшинства, такие, как исполнение  $\times$  до  $+$ .
- (2) Правило, что правый аргумент одноместной функции является значением всего выражения, находящегося справа от него, подразумеваемое в порядке исполнения такого выражения, как  $SIN LOG !N$ , обобщается на двуместные функции.

У второго правила имеются определенные полезные следствия применительно к сведению и развертке. Поскольку выражение  $F/V$  эквивалентно помещению функции  $F$  между элементами вектора  $V$ , то выражение  $-/V$  дает знакопеременную сумму элементов из  $V$ , а  $\div/V$  дает знакопеременное произведение. Кроме того, если  $B$  — булев вектор, то  $</B$  «выделяет» первую единицу в  $B$ , потому что все элементы, следующие за ней, становятся нулями. Например:

$$</0011011 \leftarrow \rightarrow 0010000$$

Дальнейшее упрощение синтаксических правил достигается применением единой формы для всех двуместных функций, которые помещаются между своими аргументами, и для всех одноместных функций, которые помещаются перед своими аргументами. Это единообразие резко отличается от разнообразия правил математики. Так, символы одноместных функций отрицания, факториала и модуля вектора соответственно предшествуют своим аргументам, следуют за ними и окружают их. Двуместные математические функции демонстрируют еще большее разнообразие.

## 1.5. ДОСТУПНОСТЬ ФОРМАЛЬНЫХ ДОКАЗАТЕЛЬСТВ

Важность формальных доказательств и выводов очевидным образом следует из их роли в математике. Раздел 4 в значи-

тельной степени посвящен формальным доказательствам, и здесь мы ограничим обсуждение введением в формы доказательств.

Доказательство исчерпывающим перебором состоит в систематическом исследовании всего конечного количества частных случаев. Такое исчерпывание часто можно выразить просто, применив некоторое внешнее произведение к аргументам, включающим все элементы соответствующей области. Например, если  $D \leftarrow 0 \ 1$ , то  $D \circ \wedge D$  дает все случаи применения функции  $\wedge$ . Кроме того, закон Де-Моргана может быть доказан исчерпывающим перебором с помощью сравнения каждого элемента матрицы  $D \circ \wedge D$  с каждым элементом из  $\sim(\sim D) \circ \vee(\sim D)$  следующим образом:

$$\begin{array}{ccc}
 & D \circ \wedge D & \sim(\sim D) \circ \vee(\sim D) \\
 \begin{array}{cc} 0 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 1 \end{array} & & \begin{array}{cc} 0 & 0 \\ 0 & 1 \end{array} \\
 & (D \circ \wedge D) = (\sim(\sim D) \circ \vee(\sim D)) & \\
 1 & \wedge / , (D \circ \wedge D) = (\sim(\sim D) \circ \vee(\sim D)) / & \\
 1 & & 
 \end{array}$$

С проблемами ассоциативности можно обходиться аналогичным образом; следующие выражения демонстрируют ассоциативность  $\wedge$  и неассоциативность  $\neg \wedge$ :

$$\begin{array}{ccc}
 1 & \wedge / , ((D \circ \wedge D) \circ \wedge D) = (D \circ \wedge (D \circ \wedge D)) & \\
 0 & \wedge / , ((D \circ \neg \wedge D) \circ \neg \wedge D) = (D \circ \neg \wedge (D \circ \neg \wedge D)) & 
 \end{array}$$

Доказательство последовательностью тождеств представляется перечислением последовательности выражений, причем каждое выражение аннотируется для поддержания очевидности его эквивалентности с его предшественником. Например, формальное доказательство тождества А.1, представленного в первом рассмотренном примере, было бы представлено следующим образом:

$$\begin{array}{ll}
 + / 1 N & \\
 + / \phi 1 N & + \text{ is associative and commutative} \\
 ((+ / 1 N) + (+ / \phi 1 N)) \div 2 & (X + X) \div 2 \leftrightarrow X \\
 (+ / ((1 N) + (\phi 1 N))) \div 2 & + \text{ is associative and commutative} \\
 (+ / ((N + 1) \rho N)) \div 2 & \text{Lemma} \\
 ((N + 1) \times N) \div 2 & \text{Definition of } \times
 \end{array}$$

Четвертое примечание относится к тождеству, которое (после рассмотрения примера в специальном случае  $(\iota 5) + (\phi 5)$ ) можно считать очевидным или же считать заслуживающим формального доказательства в качестве отдельной леммы.

Индуктивные доказательства проводятся в два этапа: (1) предполагается истинность некоторого тождества (называемого индуктивным предположением) для некоего фиксированного значения некоторого параметра  $N$ , и это предположение используется для доказательства справедливости этого тождества и для значения  $N+1$ , и (2) показывается, что это тождество справедливо для некоторого целого значения  $K$ . Вывод состоит в том, что тождество верно для всех целых значений  $N$ , которые равны или превосходят значение  $K$ .

Рекурсивные описания часто обеспечивают удобную основу для индуктивных доказательств. Для примера обратимся к рекурсивному описанию функции  $BC$  биномиальных коэффициентов, данному в А.3, и используем его в индуктивном доказательстве того, что сумма биномиальных коэффициентов порядка  $N$  равна  $2*N$ .

В качестве индуктивного предположения примем тождество

$$+ / BC \ N \leftarrow \rightarrow 2 * N$$

и затем предпримем следующие шаги:

$+ / BC \ N+1$	
$+ / (X, 0) + (0, X + BC \ N)$	A.3
$(+ / X, 0) + (+ / 0, X)$	+ is associative and commutative
$(+ / X) + (+ / X)$	$0 + Y \leftrightarrow Y$
$2 \times + / X$	$Y + Y \leftrightarrow 2 \times Y$
$2 \times + / BC \ N$	Definition of $X$
$2 \times 2 * N$	Induction hypothesis
$2 * N + 1$	Property of Power (*)

Остается показать, что индуктивное предположение верно для некоторого целого значения  $N$ . Согласно рекурсивному описанию А.3, значение  $BC \ 0$  представляет собой значение самого правого выражения, т.е. 1. Поэтому  $+ / BC \ 0$  равняется 1 и, следовательно, равняется  $2*0$ .

Закончим доказательством того, что закон Де-Моргана для скалярных аргументов, представленный в виде

$$A \wedge B \leftarrow \rightarrow \sim (\sim A) \vee (\sim B) \quad \text{A.4}$$

и доказываемый перебором, может на самом деле быть обобщен на векторы произвольной длины, как указывалось раньше гипотетическим тождеством

$$\wedge / V \leftarrow \rightarrow \sim \vee / \sim V \quad \text{A.5}$$

В качестве индуктивного предположения примем, что А.5 является истиной для векторов длины  $(\rho V) - 1$ .

Дадим сначала формальные рекурсивные описания выводимых функций *и-сведения* и *или-сведения* ( $\wedge$  и  $\vee$ ) с помощью двух новых примитивов, *индексации* и *сбрасывания*. Индексация обозначается выражением вида  $X[1]$ , где 1 — одиночный индекс или массив индексов для вектора  $X$ . Например, если  $X \leftarrow 2\ 3\ 5\ 7$ , то  $X[2]$  есть 3 и  $X[2\ 1]$  есть 3 2. Сбрасывание обозначается как  $K \downarrow X$  и описывается как отбрасывание  $|K|$  (т.е. модуль  $K$ ) элементов из  $X$ , начиная с головы, если  $K > 0$ , и с хвоста, если  $K < 0$ . Например,  $2 \downarrow X$  равняется 5 7 и  $-2 \downarrow X$  равняется 2 3. Функция *взятие* (которая будет применяться позднее) обозначается как  $\uparrow$  и описывается аналогичным образом. Например,  $3 \uparrow X$  есть 2 3 5 и  $-3 \uparrow X$  есть 3 5 7.

Формальные описания *и-сведения* и *или-сведения* обеспечиваются следующими функциями:

$$ANDRED: \omega[1] \wedge ANDRED\ 1 \downarrow \omega : 0 = \rho\omega : 1 \quad A.6$$

$$ORRED: \omega[1] \vee ORRED\ 1 \downarrow \omega : 0 = \rho\omega : 0 \quad A.7$$

Индуктивное доказательство тождества А.5 выполняется так:

$\wedge / V$

$$(V[1]) \wedge (\wedge / 1 + V) \quad A.6$$

$$\sim(\sim V[1]) \vee (\sim \wedge / 1 + V) \quad A.4$$

$$\sim(\sim V[1]) \vee (\sim \sim \vee / \sim 1 + V) \quad A.5$$

$$\sim(\sim V[1]) \vee (\vee / \sim 1 + V) \quad \sim \sim X \leftrightarrow X$$

$$\sim \vee / (\sim V[1]), (\sim 1 + V) \quad A.7$$

$$\sim \vee / \sim (V[1], 1 + V) \quad \vee \text{ distributes over ,}$$

$$\sim \vee / \sim V \quad \text{Definition of , (catenation)}$$

## 2. ПОЛИНОМЫ

Если  $C$  — вектор коэффициентов и  $X$  — скаляр, то полином от  $X$  с коэффициентом  $C$  может быть записан просто как  $+ / C \times X^{*-1} + \rho C$ , или  $+ / (X^{*-1} + \rho C) \times C$ , или  $(X^{*-1} + \rho C) + . \times C$ . Однако для применения к не скалярному массиву аргументов  $X$  следует заменить степенную функцию  $*$  на степенную таблицу  $\circ . *$ , как показано в следующем определении полиномиальной функции:

$$\underline{P}: (\omega \circ . *^{-1} + \rho \alpha) + . \times \alpha \quad B.1$$

Например:  $1\ 3\ 3\ 1\ \underline{P}\ 0\ 1\ 2\ 3\ 4 \leftarrow \rightarrow 1\ 8\ 27\ 64\ 125$ . Если  $\rho \alpha$  заменяется на  $1 \uparrow \rho \alpha$ , то функция применяется также к матрицам коэффициентов и массивам коэффициентов большей размерности,

представляющим (вдоль ведущей оси для  $\alpha$ ) наборы коэффициентов различных полиномов.

Из этого описания ясно видно, что полином является линейной функцией от вектора коэффициентов. Кроме того, если  $\alpha$  и  $\omega$  — это векторы одинаковой формы, то предмультипликатор  $\omega \circ *^{-1} + \iota \rho \alpha$  является матрицей Вандермонда для  $\omega$  и поэтому обратим, если элементы из  $\omega$  различны. Следовательно, если  $C$  и  $X$  — векторы одинаковой формы и если  $Y \leftarrow CPX$ , то обратная проблема (подгонки кривой) очевидным образом решается применением функции  $\left[ \begin{smallmatrix} \cdot \\ \cdot \\ \cdot \end{smallmatrix} \right]$  обращения матрицы к матрице Вандермонда и использованием тождества

$$C \leftarrow \rightarrow \left( \left[ \begin{smallmatrix} \cdot \\ \cdot \\ \cdot \end{smallmatrix} \right] X \circ *^{-1} + \iota \rho X \right) + \cdot \times Y$$

## 2.1. ПРОИЗВЕДЕНИЯ ПОЛИНОМОВ

Произведением двух полиномов  $B$  и  $C$  обычно считается вектор коэффициентов  $D$ , такой, что

$$D \underline{P} X \leftarrow \rightarrow (B \underline{P} X) \times (C \underline{P} X)$$

Хорошо известно, что можно вычислить  $D$ , взяв произведения всех пар элементов из  $B$  и  $C$  и просуммировав их по подмножествам тех произведений, которые ассоциируются с одной и той же степенью результата. Эти произведения появляются в таблице функции  $B \circ \times C$ , и легко показать неформально, что степени  $X$ , ассоциирующиеся с элементами  $B \circ \times C$ , задаются таблицей сложения  $E \leftarrow (-1 + \iota \rho B) \circ \cdot + (-1 + \iota \rho C)$ . Например:

$X+2$												
$B+3 \quad 1 \quad 2 \quad 3$												
$C+2 \quad 0 \quad 3$												
$E+(-1+\iota \rho B) \circ \cdot + (-1+\iota \rho C)$												
$B \circ \cdot \times C$						$E$		$X * E$				
6	0	9				0	1	2		1	2	4
2	0	3				1	2	3		2	4	8
4	0	6				2	3	4		4	8	16
6	0	9				3	4	5		8	16	32
$+ / , (B \circ \cdot \times C) \times X * E$												

518

$$(B \underline{P} X) \times (C \underline{P} X)$$

518

Из сказанного вытекает следующее тождество, которое будет установлено формально в разд. 4.

$$(B \underline{P} X) \times (C \underline{P} X) \leftarrow \rightarrow + / , (B \circ \cdot \times C) \times X * (-1 + \iota \rho B) \circ \cdot + (-1 + \iota \rho C)$$

B.2

Кроме того, вид экспоненциальной таблицы  $E$  показывает, что лежащие на диагоналях элементы из  $B \circ \times C$  ассоциируются с одной и той же степенью и что поэтому вектор коэффициентов полинома-произведения задается суммами по этим диагоналям. Иначе говоря, таблица  $B \circ \times C$  обеспечивает отличную организацию ручного вычисления произведений полиномов. В нашем примере эти суммы дают вектор  $D \leftarrow 6 \ 2 \ 13 \ 9 \ 6 \ 9$ , и можно увидеть, что  $\underline{D} \ \underline{P} \ X$  равняется  $(\underline{B} \ \underline{P} \ X) \times (\underline{C} \ \underline{P} \ X)$

Суммы по нужным диагоналям матрицы  $B \circ \times C$  можно получить также, окаймив эту матрицу нулями, «сдвинув» полученную матрицу циклической перестановкой (сдвигом) последовательных строк на число позиций, равное последовательным целым, и затем просуммировав столбцы. Таким образом, мы получаем описание произведения полиномов в следующем виде:

$$PP: + \uparrow (1 - \uparrow \rho \alpha) \phi \alpha \circ \times \omega, 1 \downarrow 0 \times \alpha$$

Теперь мы разработаем иной метод, основанный на простом наблюдении, что если  $B \ PP \ C$  порождает произведение полиномов  $B$  и  $C$ , то функция  $PP$  линейна по обоим своим аргументам. Следовательно,

$$PP: \alpha + \cdot \times A + \cdot \times \omega$$

где  $A$  — это массив, который нужно определить. Массив  $A$  должен иметь ранг 3 и зависеть от экспонент левого аргумента  $(-1 + \uparrow \rho \alpha)$ , результата  $(-1 + \uparrow \rho 1 \downarrow \alpha, \omega)$  и правого аргумента. «Дефицит» правой экспоненты задан таблицей разностей  $(\uparrow \rho 1 \downarrow \alpha, \omega) \circ - \uparrow \rho \omega$ , и сравнение этих значений с левыми экспонентами порождает  $A$ . Итак,

$$A \leftarrow (-1 + \uparrow \rho \alpha) \circ = ((\uparrow \rho 1 \downarrow \alpha, \omega) \circ - \uparrow \rho \omega)$$

и

$$PP: \alpha + \cdot \times ((-1 + \uparrow \rho \alpha) \circ = (\uparrow \rho 1 \downarrow \alpha, \omega) \circ - \uparrow \rho \omega) + \cdot \times \omega$$

Поскольку  $\alpha + \cdot \times A$  является матрицей, из этой формулы следует, что если  $D \leftarrow B \ PP \ C$ , то можно получить  $C$  из  $D$ , предварительно умножив  $D$  на обратную матрицу  $(\left[ \frac{\cdot}{\cdot} \right] B + \cdot \times A)$ ; тем самым обеспечивается деление полиномов. Так как матрица  $B + \cdot \times A$  не квадратная (в ней больше строк, чем столбцов), то этот способ не годится, но если заменить матрицу  $M \leftarrow \leftarrow B + \cdot \times A$  либо ее головной квадратной частью  $(2\rho \downarrow / \rho M) \uparrow M$ , либо хвостовой квадратной частью  $(-2\rho \downarrow / \rho M) \uparrow M$ , то получаются два результата, один из которых соответствует делению с остаточными членами нижнего порядка, а другой соответствует делению с остаточными членами более высокого порядка.

## 2.2. ПРОИЗВОДНАЯ ПОЛИНОМА

Так как производная от  $X \times N$  равняется  $N \times X \cdot N - 1$ , мы можем применить правила дифференцирования суммы функций и произведения функции на константу, чтобы показать, что производная полинома  $C \underline{P} X$  является полиномом  $(1 \downarrow C \times -1 + \downarrow C) \underline{P} X$ . Из этого результата ясно, что интеграл представляет собой полином  $(A, C \div \downarrow C) \underline{P} X$ , где  $A$  — произвольная скалярная константа. Выражение  $1 \downarrow C \times -1 + \downarrow C$  порождает также коэффициенты производной в виде вектора такой же формы, как  $C$ , с последним элементом, равным нулю.

## 2.3. ПРОИЗВОДНАЯ ПОЛИНОМА ПО ОТНОШЕНИЮ К ЕГО КОРНЯМ

Если  $R$  — вектор из трех элементов, то производные от полинома  $X/X - R$  по отношению к каждому из его корней равны  $-(X - R[2]) \times (X - R[3])$ ,  $-(X - R[1]) \times (X - R[3])$  и  $-(X - R[1]) \times (X - R[2])$ . В более общем случае производная от  $X/X - R$  по отношению к  $R[J]$  равна просто  $-(X - R) \times . * J \neq \downarrow R$ , а вектор производных относительно каждого из корней равен  $-(X - R) \times . * I_0 \neq I \leftarrow \downarrow R$ .

Выражение  $X/X - R$  для полинома с корнями  $R$  применимо только к скаляру  $X$ , более общее выражение имеет вид  $X/X_0 - D$ . Поэтому общее выражение для матрицы производных (полинома, вычисленного в  $X[I]$  по отношению к корню  $R[J]$ ) задается так:

$$-(X_0 - R) \times . * I_0 \neq I \leftarrow \downarrow R \quad \text{B.3}$$

## 2.4. РАЗЛОЖЕНИЕ ПОЛИНОМА

Биномиальное разложение — это выписывание тождества в форме полинома от  $X$  для выражения  $(X + Y) * N$ . Для частного случая  $Y = 1$  мы имеем хорошо известное выражение в терминах биномиальных коэффициентов порядка  $N$ :

$$(X + 1) * N \leftarrow \rightarrow ((0, 1N)!N) !N) \underline{P} X$$

В общем случае коэффициенты  $D$  являются функциями от  $Y$ . Если  $Y = 1$ , они опять зависят только от биномиальных коэффициентов, но в этом случае от нескольких биномиальных коэффициентов разных порядков, конкретно от матрицы  $J_0 ! J \leftarrow -1 + \downarrow C$ .

Например, если  $C \leftarrow 3 \ 1 \ 2 \ 4$  и  $C \underline{P} X + 1 \leftarrow \rightarrow D \underline{P} X$ , то  $D$  зависит от матрицы



	0	1	2	3	...	0	1	2	3
1	1	1	1	1					
0	1	2	3						
0	0	1	3						
0	0	0	1						

и ясно, что величина  $D$  должна быть взвешенной суммой столбцов, причем весами являются элементы из  $C$ . Итак:

$$D \leftarrow (J \circ ! J \leftarrow^{-1} + \iota \rho C) + . \times C$$

Краткая запись матрицы коэффициентов и выполнение указанного перемножения матриц обеспечивают быстрый и надежный способ организации громоздкого без этих приемов ручного вычисления разложений.

Если  $B$  — подходящая матрица биномиальных коэффициентов, то ясно, что  $D \leftarrow B + . \times C$  и функция разложения линейна относительно коэффициентов  $C$ . Кроме того, разложение для  $Y = -1$  должно задаваться обратной матрицей  $\left[ \frac{\cdot}{\cdot} \right] B$ , которая, как мы увидим, содержит знакопеременные биномиальные коэффициенты. Наконец, поскольку

$$C \underline{P} X + (K + 1) \leftarrow \rightarrow C \underline{P} (X + K) + 1 \leftarrow \rightarrow (B + . \times C) \underline{P} (X + K).$$

то, следовательно, разложение для положительных целых значений  $Y$  можно задать произведениями вида

$$B + . \times B + . \times B + . \times B + . \times C$$

где  $B$  появляется  $Y$  раз.

Так как суперпозиция  $+ . \times$  ассоциативная, то предыдущее выражение можно записать в виде  $M + . \times C$ , где  $M$  является произведением  $Y$  появлений  $B$ . Интересно исследовать последовательные степени  $B$ , вычисленные либо вручную, либо выполнением на компьютере следующей степенной функции внутреннего произведения ( $IPP$ ):

$$IPP: \alpha + . \times \alpha \quad IPP \omega - 1 : \omega = 0 : J \circ . = J \leftarrow^{-1} + \iota 1 \uparrow \rho \alpha$$

Сравнение  $B \text{ IPP } K$  с  $B$  для нескольких значений  $K$  показывает очевидную закономерность, которая может быть выражена в следующем виде:

$$B \text{ IPP } K \leftarrow \rightarrow B \times K * 0 \vdash - J \circ . - J \leftarrow^{-1} + \iota 1 \uparrow \rho B$$

Интересно, что правая часть этого тождества имеет смысл для нецелых значений  $K$  и фактически обеспечивает желаемое выражение для разложения  $C \underline{P} X + Y$ :

$$C \underline{P} (X + Y) \leftarrow \rightarrow$$

$$(((J_0. ! J) \times Y * 0 \sqcap - J_0. - J \leftarrow^{-1} + \uparrow \rho C) + . \times C) \underline{P} X \quad B.4$$

Правая часть из B.4 имеет вид  $(M + . \times C) \underline{P} X$ , где матрица  $M$  в свою очередь имеет вид  $B \times Y * E$  и может быть изображена неформально (для случая  $4 = \rho C$ ) следующим образом:

$$\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{array} \quad \times Y * \quad \begin{array}{cccc} 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}$$

Поскольку  $Y * K$  умножено на диагональную матрицу  $B \times (K = E)$ , то выражение для  $M$  можно также записать как внутреннее произведение  $(Y * J) + . \times T$ , где  $T$  — массив ранга 3 (т. е. трехмерный), в котором плоскость с номером  $K$  является матрицей  $B \times (K = E)$ . Такой массив ранга 3 можно сформировать из верхней треугольной матрицы  $M$ , образовав массив ранга 3, в котором первая плоскость — это  $M$  (т. е.  $(1 = \uparrow \rho M) \circ . \times M$ ), и поворачивая его вдоль первой оси с помощью матрицы  $J_0. - J$ , для которой супердиагональ с номером  $K$  имеет значение  $-K$ . Итак:

$$DS : (I_0. - I) \Phi[1] (1 = I + \uparrow \rho \omega) \circ . \wedge \omega \quad B.5$$

$$DS \quad K_0. ! K \leftarrow^{-1} + \uparrow \rho$$

$$\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}$$

$$\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{array}$$

$$\begin{array}{ccc} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}$$

Подставив этот результат в B.4 и воспользовавшись ассоциативностью суперпозиции  $+ . \times$ , мы получаем следующее тождество для разложения полинома, справедливое как для целых, так и для нецелых значений  $Y$ :

$$C \underline{P} X + Y \leftarrow \rightarrow ((Y * J) + . \times (DS \ J_0. ! J) \leftarrow^{-1} + \uparrow \rho C) + . \times C) \underline{P} X \quad B.6$$

Например:

$$\begin{array}{l}
 Y \leftarrow 3 \\
 C \leftarrow 3 \quad 1 \quad 4 \quad 2 \\
 M \leftarrow (Y * J) + . \times DS \quad J \circ . ! J \leftarrow 1 + 1 \rho C \\
 M \\
 \begin{array}{cccc}
 1 & 3 & 9 & 27 \\
 0 & 1 & 6 & 27 \\
 0 & 0 & 1 & 9 \\
 0 & 0 & 0 & 1
 \end{array} \\
 M + . \times C \\
 96 \quad 79 \quad 22 \quad 2 \\
 (M + . \times C) \underline{P} \quad X \leftarrow 2 \\
 358 \\
 C \underline{P} \quad X + Y \\
 358
 \end{array}$$

### 3. ПРЕДСТАВЛЕНИЯ

Объекты математического анализа и вычислений могут быть *представлены* разнообразными способами, и каждое представление может обладать конкретными преимуществами. Например, положительное число  $N$  можно представить просто  $N$  палочками, или менее просто, но более компактно посредством римских цифр, или же менее просто, но более удобно для выполнения сложения и умножения в десятичной системе, или менее привычно, но более удобно для вычисления наименьшего общего кратного и наибольшего общего делителя в схеме разложения на простые множители, которая будет здесь обсуждаться.

Графы, относящиеся к связям между набором элементов, представляют собой пример более сложного объекта, которому соответствует несколько полезных представлений. Так, простой ориентированный граф из  $N$  элементов (обычно называемых *узлами*) можно представить булевой матрицей  $B$  размера  $N \times N$  (обычно называемой матрицей *смежности*), такой, что  $B[I; J] = 1$ , если имеется связь от вершины  $I$  к вершине  $J$ . Всякая связь, представленная единицей в матрице  $B$ , называется *дугой*, и граф может быть представлен также матрицей из  $+/,-$ ,  $B$  строк и  $N$  столбцов, в которой каждая строка показывает узлы, связываемые конкретной дугой.

Функции также допускают различные полезные представления. Например, функция перестановки, порождающая переупорядочивание элементов своего векторного аргумента  $X$ , может быть представлена вектором перестановки  $P$ , таким, что функция перестановки имеет вид просто  $X[P]$ ; *циклическим* представлением, которое более явно показывает структуру этой

функции; булевой матрицей  $B \leftarrow P = {}_{\rho}P$ , такой, что функция перестановки — это  $B + \times X$ ; или же *корневым* представлением  $R$ , которое основывается на одном из столбцов матрицы  $1 + (\phi_1 N) \top - 1 + {}_{\rho}N \leftarrow {}_{\rho}X$  и обладает тем свойством, что  $2 \mid + / R - 1$  является четностью представляемой перестановки.

Для того чтобы удобным образом использовать различные представления, важно уметь ясно и точно выражать преобразования между представлениями. Обычная математическая нотация часто оказывается недостаточной в этом отношении, и настоящий раздел посвящен разработке выражений для преобразований между представлениями, полезными в разнообразных приложениях: в системах счисления, полиномах, перестановках, теории графов и в булевой алгебре.

### 3.1. СИСТЕМЫ СЧИСЛЕНИЯ

Начнем обсуждение представлений с уже знакомого примера, применения различных представлений положительных чисел и преобразований между ними. Вместо обычно употребляемого *позиционного* представления мы будем использовать *разложение на простые множители*, т.е. представление, интересные свойства которого делают его полезным и для введения понятия логарифмов, и для представления чисел [6, гл. 16].

Если  $P$  — вектор из первых  $\rho P$  простых чисел и  $E$  — вектор неотрицательных целых, то  $E$  может служить для представления числа  $P \times . * E$ , и аналогичным образом можно представить любые целые из  $\top / P$ . Например,  $2\ 3\ 5\ 7 \times . * 0\ 0\ 0\ 0$  равняется 1, а  $2\ 3\ 5\ 7 \times . * 1\ 1\ 0\ 0$  равняется 6, а также

$P$									
2	3	5	7						
$ME$									
0	1	0	2	0	1	0	3	0	1
0	0	1	0	0	1	0	0	2	0
0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	1	0	0	0
$P \times . * ME$									
1	2	3	4	5	6	7	8	9	10

Сходство с логарифмами видно из тождества

$$\times / P \times . * ME \leftarrow \rightarrow P \times . * + / ME$$

которое может служить для реализации умножения сложением.

Кроме того, если мы обозначим через  $GCD$  и  $LCM$  наибольший общий делитель и наименьшее общее кратное элементов векторных аргументов, то

$$GCD P \times . * ME \leftrightarrow P \times . * \downarrow / ME$$

$$LCM P \times . * ME \leftrightarrow P \times . * \uparrow / ME$$

$ME$			$V \leftarrow P \times . * ME$		
2	1	0	$V$		
3	1	2	18900	7350	3087
2	2	0	$GCD V$		
1	2	3	21	$LCM V$	
				926100	
			$P \times . * \downarrow / ME$		$P \times . * \uparrow / ME$
			21	926100	

При описании функции  $GCD$  мы воспользуемся операцией / с булевым аргументом  $B$  [как в  $B/$ ]. Получается функция *сжатия*, которая выбирает элементы из своего правого аргумента в соответствии с единицами в  $B$ . Например,  $10101/15$  равняется 135. Кроме того, функция  $B/$ , примененная к матричному аргументу, сжимает строки (выбирая определенные столбцы), а функция  $B\uparrow$  сжимает столбцы для выбора строк. Итак:

$$GCD: GCD M, (M \uparrow \downarrow / R) \downarrow R: 1 \geq \rho R \leftarrow (\omega \neq 0) / \omega: + / R$$

$$LCM: (\times / X) \div GCD X \leftarrow (1 + \omega), LCM 1 + \omega: 0 = \rho \omega: 1$$

Преобразование в значение числа из его представления в виде разложения на простые множители ( $VFR$ ) и обратное преобразование в это представление из значения ( $RFV$ ) задаются так:

$$VFR: \alpha \times . * \omega$$

$$RFV: D + \alpha \quad RFV \quad \omega \div \alpha \times . * D: \wedge / \sim D + 0 = \alpha \mid \omega: D$$

Например:

	$P$	$VFR$	2	1	3	1
10500						
	$P$	$RFV$	10500			
2	1	3	1			

### 3.2. ПОЛИНОМЫ

В разд. 2 введены два представления полинома от скалярного аргумента  $X$ , первое в терминах вектора коэффициентов  $C$  (т.е.  $+ / C \times X^{*-1} + \uparrow C$ ), а второе в терминах его корней  $R$  (т.е.  $\times / X - R$ ). Коэффициентное представление удобно для

сложения полиномов  $(C+D)$  и для получения производных  $(1\downarrow C \times -1 + \uparrow \rho C)$ . Корневое представление удобно для других целей, в том числе для умножения, которое задается посредством  $R_1, R_2$ .

Теперь мы разработаем функцию  $CFR$  (коэффициенты из корней), которая преобразует корневое представление в эквивалентное представление, и обратную функцию  $RFC$ . Разработка будет неформальной; формальный вывод  $CFR$  появится в разд. 4.

Выражение для  $CFR$  будет основываться на симметричных функциях Ньютона, которые порождают коэффициенты как суммы по некоторым из произведений над всеми подмножествами арифметического отрицания (т.е.  $-R$ ) корней  $R$ . Например, коэффициент константного члена задается как  $\times / -R$ , т.е. как произведение над всем множеством, а коэффициент при следующем члене равняется сумме произведений над элементами из  $-R$ , взятыми по  $(\rho R) - 1$  каждый раз.

Функция, описанная в А.2, может служить для получения произведений над всеми подмножествами следующим образом:

$$P \leftarrow (-R) \times . * M \leftarrow \underline{T} \rho R$$

Элементы из  $P$ , суммированные для получения заданного коэффициента, зависят от числа элементов из  $R$ , исключенных из конкретного произведения, т.е. от  $+f \sim M$ , суммы столбцов дополнения булевой матрицы «подмножества»  $\underline{T} \rho R$ .

Таким образом, суммирование над  $P$  может быть выражено как  $((0, \uparrow \rho R) \circ . = +f \sim M) + . \times P$ , и полное выражение для коэффициентов  $C$  приобретает вид

$$C \leftarrow ((0, \uparrow \rho R) \circ . = +f \sim M) + . \times (-R) \times . * M \leftarrow \underline{T} \rho R$$

Например, если  $R \leftarrow 2 \ 3 \ 5$ , то

$M$	$+f \sim M$
0 0 0 0 1 1 1 1	3 2 2 1 2 1 1 0
0 0 1 1 0 0 1 1	$(0, \uparrow \rho R) \circ . = +f \sim M$
0 1 0 1 0 1 0 1	0 0 0 0 0 0 0 1
$(-R) \times . * M$	0 0 0 1 0 1 1 0
1 -5 -3 15 -2 10 6 -30	0 1 1 0 1 0 0 0
	1 0 0 0 0 0 0 0
$((0, \uparrow \rho R) \circ . = +f \sim M) + . \times (-R) \times . * M \leftarrow \underline{T} \rho R$	
-30 31 -10 1	

Поэтому функция  $CFR$  получения коэффициентов из корней может быть описана и использована следующим образом:

$$CFR:((0, \rho\omega) \circ \cdot + \sim M) + \cdot \times (-\omega) \times \cdot * M + \tau \rho\omega \quad C.1$$

$$\begin{array}{r} CFR \quad 2 \quad 3 \quad 5 \\ -30 \quad 31 \quad -10 \quad 1 \\ (CFR \quad 2 \quad 3 \quad 5) \quad \underline{P} \quad X+1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \\ -8 \quad 0 \quad 0 \quad -2 \quad 0 \quad 12 \quad 40 \quad 90 \\ \times / X \circ \cdot -2 \quad 3 \quad 5 \\ -8 \quad 0 \quad 0 \quad -2 \quad 0 \quad 12 \quad 40 \quad 90 \end{array}$$

Обратное преобразование  $RFC$  является более трудным, но может быть выражено как схема последовательных приближений:

$$\begin{array}{l} RFC: (-1 + \rho 1 + \omega) G \quad \omega \\ G: (\alpha - Z) G \quad \omega: TOL \geq \lceil / / Z + \alpha \quad STEP \quad \omega: \alpha - Z \\ STEP: (\mathbb{B}(\alpha \circ \cdot - \alpha) \times \cdot * I \circ \cdot \neq I + \rho \alpha) + \cdot \times (\alpha \circ \cdot * -1 + \rho \omega) + \cdot \times \omega \\ \square + C + CFR \quad 2 \quad 3 \quad 5 \quad 7 \\ 210 \quad -247 \quad 101 \quad -17 \quad 1 \\ TOL + 1E^{-8} \\ RFC \quad C \\ 7 \quad 5 \quad 2 \quad 3 \end{array}$$

Разумеется, порядок корней в результате не имеет значения. Последним элементом любого аргумента для  $RFC$  должна быть единица, поскольку любой полином, эквивалентный  $X/X - R$ , обязательно должен иметь коэффициент 1 для члена самого высокого порядка.

Данное выше описание  $RFC$  применимо только для коэффициентов полиномов, все корни которых являются вещественными. Левый аргумент для  $G$  в  $RFC$  обеспечивает (обычно удовлетворительные) начальные приближения для корней, но в общем случае по крайней мере некоторые из них должны быть комплексными. Следующий пример с использованием единицы как начального приближения корней был выполнен в системе APL, оперирующей комплексными числами:

$$\begin{array}{l} (*00J2 \times (-1 + \rho N) + N + \rho 1 + \omega) G \omega \quad C.2 \\ \square + C + CFR \quad 1J1 \quad 1J^{-1} \quad 1J2 \quad 1J^{-2} \\ 10 \quad -14 \quad 11 \quad -4 \quad 1 \\ RFC \quad C \\ 1J^{-1} \quad 1J2 \quad 1J1 \quad 1J^{-2} \end{array}$$

Использованная выше одноместная функция  $\circ$  умножает свой аргумент на  $\pi$ .

В методе Ньютона отыскания корня скалярной функции  $F$  следующее приближение задается как  $A \leftarrow A - (FA) \div DFA$ , где  $DF$  — производная от  $F$ . Функция  $STEP$  представляет собой обобщение метода Ньютона на случай, когда  $F$  является векторной функцией от вектора. Она имеет вид  $(\lfloor \cdot \rfloor M) + \times B$ , где  $B$  — это значение полинома с коэффициентами  $\omega$ ;  $\omega$  — исходный аргумент для  $RFC$ , вычисленный в  $\alpha$ , т. е. в текущем приближении для корней. Анализ, подобный выводу формулы В.3, показывает, что  $M$  является матрицей производных полинома с корнями  $\alpha$ , причем производные вычисляются в  $\alpha$ .

Исследование выражения для  $M$  показывает, что все внедиагональные элементы этой матрицы представляют собой нули, и поэтому выражение  $(\lfloor \cdot \rfloor M) + \times B$  можно заменить на  $B \div D$ , где  $D$  — это вектор диагональных элементов из  $M$ . Поскольку операция  $(I, J) \downarrow N$  отбрасывает  $I$  строк и  $J$  столбцов из матрицы  $N$ , то вектор  $D$  можно выразить как  $\times / 0 \ 1 \downarrow (-1 + \uparrow \alpha) \phi \alpha \cdot - \alpha$ ; поэтому описание функции  $STEP$  можно заменить на более эффективное описание:

$$STEP : ((\alpha \cdot * -1 + \uparrow \rho \omega) + \times \omega) \div 0 \ 1 \downarrow (-1 + \uparrow \alpha) \phi \alpha \cdot - \alpha \quad C.3$$

Последняя формула реализует изящный метод Кернера [7]. С помощью начальных значений, задаваемых левым аргументом для  $G$  в С.2, эта аппроксимация сходится за семь шагов (с отклонением  $TOL \leftarrow 1E-8$ ) для представленного Кернером примера шестого порядка.

### 3.3. ПЕРЕСТАНОВКИ

Вектор  $P$ , элементы которого представляют собой некоторую перестановку его индексов (т. е.  $\wedge / 1 = + / P \cdot + \uparrow \rho P$ ), будет называться вектором *перестановок*. Если  $D$  — такой вектор перестановок, что  $(\rho X) = \rho D$ , то  $X[D]$  является перестановкой от  $X$  и  $D$  будет называться *прямым представлением* этой перестановки.

Перестановка  $X[D]$  может быть также выражена как  $B + \times X$ , где  $B$  — булева матрица  $D \cdot = \uparrow \rho D$ . Матрица  $B$  будет называться булевым представлением перестановки. Преобразования между прямым и булевым представлениями выглядят так:

$$BFD : \omega \cdot = \uparrow \rho \omega \quad DFB : \omega + \cdot \times \uparrow 1 \uparrow \rho \omega$$

В силу ассоциативности перестановки композиция перестановок удовлетворяет следующим отношениям:



$$(X[D1])[D2] \leftarrow \rightarrow X[(D1[D2])]$$

$$B2 + . \times (B1 + . \times X) \leftarrow \rightarrow (B2 + . \times B1) + . \times X$$

Обратным для булева представления  $B$  является  $\ominus B$ , а для прямого представления обратное — либо  $\uparrow D$ , либо  $\text{Dir} D$ . (Функция  $\uparrow$  *ранжирования* ранжирует свой аргумент, присваивая его элементам вектор индексов в восходящем порядке и сохраняя существующий порядок для равных элементов. Итак,  $\uparrow 3714$  равняется  $3142$  и  $\uparrow 3734$  равняется  $1342$ . Функция  $\downarrow$  *индекс* определяет наименьший индекс в своем левом аргументе для каждого элемента из своего правого аргумента. Например,  $'ABCDE'\downarrow'BABE'$  равняется  $2125$ , и  $'BABE'\downarrow'ABCDE'$  равняется  $21554$ .)

В *циклическом* представлении также используется вектор перестановки. Рассмотрим вектор перестановки  $C$  и сегменты из  $C$ , выделенные вектором  $C = \mathbb{L} \setminus C$ . Например, если  $C \leftarrow 7365214$ , то  $C = \mathbb{L} \setminus C$  равняется  $1100110$  и блоки имеют вид

```

7
3 6 5
2
1 4

```

Каждый блок определяет «цикл» в соответствующей перестановке в том смысле, что если  $R$  является результатом перестановки  $X$ , то

```

R[7] является X[7]
R[3] является X[6] R[6] является X[5] R[5] является X[3]
R[2] является X[2]
R[1] является X[4] R[4] является X[1]

```

Если ведущий элемент из  $C$  является наименьшим (т.е. 1), то  $C$  состоит из единственного цикла и предоставляемая перестановка вектора  $X$  задается как  $X[C] \leftarrow X[1\phi C]$ . Например:

```

X ← 'ABCDEFGF'
C ← 1 7 6 5 2 4 3
X[C] ← X[1ϕC]
X
GDACBEF

```

Поскольку эквивалентны формулы  $X[Q] \leftarrow A$  и  $X \leftarrow A[\uparrow Q]$ , то, следовательно, эквивалентны также  $X[C] \leftarrow X[1\phi C]$  и  $X \leftarrow X[(1\phi C)[\uparrow C]]$ , и поэтому эквивалентный  $C$  вектор прямого

представления  $D$  задается (для специального случая единственного цикла) как  $D \leftarrow (1\phi C)[\uparrow C]$ .

В более общем случае поворот полного вектора (т. е.  $1\phi C$ ) должен быть заменен поворотами отдельных подциклов, выделенных посредством  $C \setminus C$ , как показано в следующем описании преобразования из циклического представления в прямое:

$$DFC: (\omega [\uparrow X + + \setminus X \leftarrow \omega = \setminus / \omega]) [\uparrow \omega]$$

Если есть желание соединить ряд несвязанных циклов, чтобы сформировать единый вектор  $C$ , такой, что выражение  $C = \setminus C$  выделяет отдельные циклы, то каждый цикл  $C1$  нужно сначала привести в *стандартный вид* поворотом  $(-1 + + C1 \setminus C1)\phi C1$ , а полученные векторы затем соединить в нисходящем порядке их ведущих элементов.

Обратное преобразование из прямого представления в циклическое является более сложным, но к нему можно подступиться, построив матрицу всех степеней от  $D$  вплоть до степени  $\rho D$ , т. е. матрицу, последовательными столбцами которой являются  $D, D[D], (D[D])[D]$  и т. д. Этот результат получается применением функции  $POW$  к одностолбцовой матрице  $D^\circ. +, 0$ , сформированной из  $D$ , где  $POW$  описывается и используется следующим образом:

$$POW: POW D, (D + \omega [ ; 1 ]) (\omega) : \leq / \rho \omega : \omega$$

$$\square \leftarrow D \leftarrow DFC C + 7, 3 \ 6 \ 5, 2, 1 \ 4$$

$$4 \ 2 \ 6 \ 1 \ 3 \ 5 \ 7$$

$$POW D^\circ. +, 0$$

$$4 \ 1 \ 4 \ 1 \ 4 \ 1 \ 4$$

$$2 \ 2 \ 2 \ 2 \ 2 \ 2 \ 2$$

$$6 \ 5 \ 3 \ 6 \ 5 \ 3 \ 6$$

$$1 \ 4 \ 1 \ 4 \ 1 \ 4 \ 1$$

$$3 \ 6 \ 5 \ 3 \ 6 \ 5 \ 3$$

$$5 \ 3 \ 6 \ 5 \ 3 \ 6 \ 5$$

$$7 \ 7 \ 7 \ 7 \ 7 \ 7 \ 7$$

Если  $M \leftarrow POW D^\circ. +, 0$ , то циклическое представление можно получить выбором из  $M$  только «стандартных» строк, которые начинаются со своих наименьших элементов ( $SSR$ ), упорядочивая эти остающиеся строки в нисходящем порядке их ведущих элементов ( $DOL$ ) и последующим соединением циклов по этим строкам ( $CIR$ ). Итак:

$$CFD: CIR \ DOL \ SSR \ POW \ \omega \circ \cdot +, 0$$

$$SSR: (\wedge / M = 1 \phi M \wedge \lfloor \backslash \omega) / \omega$$

$$DOL: \omega [\Psi \omega [; 1]; ]$$

$$CIR: (, 1, \wedge \backslash 0 \ 1 + \omega \neq \lfloor \backslash \omega) / , \omega$$

$$DFC \ C \leftarrow 7, 3 \ 6 \ 5, 2, 1 \ 4$$

$$4 \ 2 \ 6 \ 1 \ 3 \ 5 \ 7$$

$$CFD \ DFC \ C$$

$$7 \ 3 \ 6 \ 5 \ 2 \ 1 \ 4$$

В описании *DOL* индексация применяется к матрицам. Индексы для последовательных координат разделяются точками с запятыми, и пустая запись для каждой оси означает, что все элементы вдоль этой оси выбраны. Так,  $M[; 1]$  выбирает столбец 1 из  $M$ .

Циклическое представление удобно для определения числа циклов в данной перестановке ( $NC: +/\omega = \lfloor \backslash \omega$ ), длин циклов ( $CL: X \leftarrow 0, -1 \downarrow X \leftarrow (1 \phi \omega = \lfloor \backslash \omega) / \omega$ ) и степени перестановки ( $PP: LCM \ CL \ \omega$ ). С другой стороны, такое представление неудобно для композиции и обращения.

Все  $!N$  векторов-столбцов из матрицы  $(\phi !N) \top^{-1} + !N$  различны, и поэтому они обеспечивают возможное *корневое* представление [8] для  $!N$  перестановок порядка  $N$ . Вместо этого мы будем использовать связанную с ним форму, получаемую увеличением каждого элемента на 1:

$$RR: 1 + (\phi !\omega) \top^{-1} + !\omega$$

$$RR \ 4$$

1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4
1	1	2	2	3	3	1	1	2	2	3	3	1	1	2	2	3	3	1	1	2	2	3	3
1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Преобразования между этим дополнительным представлением и прямой формой задаются так:

$$DFR: \omega [1], X + \omega [1] \leq X \leftarrow DFR \ 1 \downarrow \omega : 0 = \rho \omega : \omega$$

$$RFD: \omega [1], RFD \ X - \omega [1] \leq X \leftarrow 1 \downarrow \omega : 0 = \rho \omega : \omega$$

Некоторые характеристики этого варианта представления, возможно, лучше всего выявляются модификацией *DFR* для применения ко всем столбцам матричного аргумента и применением модифицированной функции *MF* к результату функции *RR*.

$$MF: \omega[., 1; ], [1]X + \omega[(1 \rho X) \rho 1; ] \leq X + MF \quad 1 \quad 0 + \omega: 0 = 1 + \rho \omega: \omega$$

$$MF \quad RR \quad 4$$

1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4
2	2	3	3	4	4	1	1	3	3	4	4	1	1	2	2	4	4	1	1	2	2	3	3
3	4	2	4	2	3	3	4	1	4	1	3	2	4	1	4	1	2	2	3	1	3	1	2
4	3	4	2	3	2	4	3	4	1	3	1	4	2	4	1	2	1	3	2	3	1	2	1

Прямые перестановки в столбцах этого результата появляются в *лексикографическом* порядке (т.е. по возрастанию первого элемента, в котором два вектора отличаются); этот факт справедлив и в общем случае, и поэтому дополнительное представление обеспечивает удобный способ получения прямых представлений в лексикографическом порядке.

Дополнительное представление имеет еще одно полезное свойство, состоящее в том, что четность прямой перестановки  $D$  задается как  $2 \mid + / - 1 + RFD D$ , где  $M \mid N$  представляет остаток от  $N$  по модулю  $M$ . Четность прямого представления может быть определена также функцией

$$PAR: 2 \mid + / . (I \circ . > I \leftarrow \rho \omega) \wedge \omega \circ . > \omega$$

### 3.4. ОРИЕНТИРОВАННЫЕ ГРАФЫ

Простой ориентированный граф описывается множеством из  $K$  узлов и множеством ориентированных соединений от одной вершины к другой для пар вершин. Ориентированные соединения удобно представляются булевой матрицей *соединений*  $C$  размера  $K \times K$ , в которой значение  $C[I; J] = 1$  обозначает соединение, ведущее из узла  $I$  в узел  $J$ .

Например, если четыре узла графа представляются как  $N \rightarrow 'QRST'$  и если имеются соединения из узла  $S$  в узел  $Q$ , из  $R$  в  $T$  и из  $T$  в  $Q$ , то соответствующая матрица соединений задается так:

0	0	0	0
0	0	0	1
1	0	0	0
1	0	0	0

Соединение от некоего узла к нему самому (называемое нулевой петлей) недопустимо, и поэтому диагональ матрицы соединений должна состоять из нулей.

Если  $P$  — это любой вектор перестановки порядка  $\rho N$ , то  $N1 \leftarrow N[P]$  является переупорядочиванием узлов, и соответ-

вующая матрица соединений задается как  $C[P; P]$ . Мы можем (и должны) без потери общности употреблять для узлов числовые метки  $\iota P N$ , потому что если  $N$  — это любой произвольный вектор имен для узлов, а  $L$  — любой список числовых меток, то выражение  $Q \leftarrow N[L]$  дает соответствующий список имен, и наоборот,  $N \iota Q$  дает список числовых меток.

Матрица соединений  $C$  удобна для выражения многих полезных функций на графе. Например,  $+ / C$  задает *выходные степени* узлов,  $+ \uparrow C$  задает *входные степени*,  $+ /, C$  задает число соединений или *дуг*,  $\otimes C$  задает родственный граф с обратными направлениями дуг и  $C \vee \otimes C$  задает родственный «симметричный» или «неориентированный» граф. Кроме того, если мы используем булев вектор  $B \leftarrow \vee / (\iota P C) \circ = L$  для представления списка вершин  $L$ , то  $B \vee, \wedge C$  задает булев вектор, который представляет множество узлов, непосредственно достижимых из множества  $B$ . Поэтому  $C \vee, \wedge C$  задает соединения для путей длины два в графе  $C$  и  $C \vee C \vee, \wedge C$  задает соединения для путей длины один или два. Это рассуждение приводит к следующей функции для *транзитивного замыкания* графа, дающего все соединения путями любой длины:

$$TC : TC \quad Z : \wedge /, \omega = Z \leftarrow \omega \wedge \omega \vee, \wedge \omega : Z$$

Узел  $J$  называется *достижимым* из узла  $I$ , если  $(TC C) [I; J] = 1$ . Граф называется *сильносвязным*, если всякий узел достижим из любого узла, т. е.  $\wedge /, TC C$ .

Если  $D \leftarrow TC C$  и  $D(I; I) = 1$  для некоего узла  $I$ , то узел  $I$  *достижим* из самого себя посредством пути некоторой длины; этот путь называется *контуром*, а об узле  $I$  говорят, что он содержится в контуре.

Граф  $T$  называется *деревом*, если он не содержит контуров и его входные степени не превышают 1, т. е.  $\wedge / 1 \geq + \uparrow T$ . В дереве любой узел с входной степенью 0 называется *корнем*, и если  $K \leftarrow + / 0 = + \uparrow T$ , то  $T$  называется  $K$ -корневым деревом. Поскольку в дереве нет контуров, то значение  $K$  должно быть не меньше 1. Если не указано обратное, то обычно предполагается, что дерево является *однокорневым* (т. е.  $K=1$ ): многокорневые деревья иногда называются *лесами*.

Граф  $C$  *покрывает* граф  $D$ , если  $\wedge /, C \geq D$ . Если  $G$  — сильносвязный граф и  $T$  — это (однокорневое) дерево, то  $T$  называется *стягивающим деревом* для  $G$ , если  $G$  покрывает  $T$  и если все узлы достижимы из корня дерева  $T$ , т. е.

$$(\wedge /, G \geq T) \wedge \wedge / R \vee R \vee, \wedge TC T$$

где  $R$  — это булево представление корня  $T$ .

*Стягивающее дерево первой глубины* [9] для графа  $G$  — это стягивающее дерево, порожденное движением из корня через непосредственных «потомков» в  $G$ , причем в качестве очередного узла всегда выбирается потомок последнего узла из списка рассмотренных узлов, который еще обладает потомком вне списка. Этот относительно сложный процесс может служить для иллюстрации применения представления в виде матрицы соединений:

$$DFST: ((, 1) \circ . = K) R \omega \wedge K \circ . \vee \sim K + \alpha = 11 \uparrow \rho \omega \quad C.4$$

$$\begin{aligned} R: (C, [1] \alpha) R \omega \wedge P \circ . \vee \sim C + < \setminus U \wedge P \vee . \wedge \omega \\ : \sim \vee / P + ( < \setminus \alpha \vee . \wedge \omega \vee . \wedge U + \sim \vee \alpha ) \vee . \wedge \alpha \\ : \omega \end{aligned}$$

Воспользуемся примером графа  $G$  из [9]:

$G$												1 $DFST\ G$												
0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Функция  $DFST$  устанавливает левый аргумент рекурсии  $R$  как однострочную матрицу, представляющую корень, указанный левым аргументом из  $DFST$ , и правый аргумент как исходный граф с удаленными соединениями, ведущими в корень  $K$ . Первая строка рекурсии  $R$  показывает, что рекурсия продолжается присоединением на верх списка узлов, собранных до сих пор в левом аргументе, очередного потомка  $C$  и удалением из правого аргумента всех соединений, направленных к выбранному потомку  $C$ , за исключением одного, идущего от его родителя  $P$ . Потомок  $C$  выбран среди достижимых из выбранного родителя ( $P \vee . \wedge \omega$ ), но еще не затронутых ( $U \wedge P \vee . \wedge \omega$ ), и берется произвольным образом как первый из них ( $< \setminus U \wedge P \vee . \wedge \omega$ ).

Определения  $P$  и  $U$  показаны во второй строке.  $P$  выбирается из тех вершин, у которых имеются потомки среди незатронутых вершин ( $\omega \vee \wedge U$ ). Над ними выполняется перестановка, упорядочивающая их согласно порядку узлов для левого аргумента ( $\alpha \vee \wedge \omega \vee \wedge U$ ), причем они обретают такой порядок, что последний затронутый узел появляется первым, и, наконец, выбирается узел  $P$  как первый из них.

Последняя строка из  $R$  показывает, что окончательный результат является новым правым аргументом  $\omega$ , т.е. исходным графом, в котором все соединения, ведущие в каждый узел, вычеркнуты, кроме соединения с его родителем в стягивающем дереве. Поскольку окончательным значением  $\alpha$  является квадратная матрица, задающая узлы дерева в порядке, обратном порядку их посещения, то подстановка  $\omega, \Phi[1]\alpha$  (или, что эквивалентно,  $\omega, \Theta\alpha$ ) для  $\omega$  породила бы результат вида  $1\ 2 \times \rho G$ , содержащий стягивающее дерево, за которым следует информация о его «предварительном упорядочении».

Часто используется, по крайней мере неявно, другое представление ориентированных графов в виде списка пар узлов  $V, W$ , таких, что имеется связь из  $V$  в  $W$ . Преобразование матрицы соединений в форму такого списка можно описать и использовать следующим образом:

$$LFC: (, \omega) / 1 + D\tau^{-1} + 1 \times / D + \rho\omega$$

$C$	$LFC\ C$
0 0 1 1	1 1 2 3 3 4
0 0 1 0	3 4 3 2 4 1
0 1 0 1	
1 0 0 0	

Однако это представление является недостаточным, поскольку само по себе оно не определяет число узлов в графе, хотя для данного примера оно задается как  $/, LFC\ C$ , потому что узел с наибольшим номером случайно имеет соединение. Соответствующее булево представление обеспечивается выражением  $(LFC\ C) \circ = 1 \uparrow \rho C$ , причем первая плоскость показывает исходящие, а вторая — заходящие соединения.

Представление в виде матрицы *инцидентности*, часто используемое при изучении электрических сетей [10], задается как разность этих плоскостей следующим образом:

$$IFC: = \uparrow (LFC\omega) \circ = 1 \uparrow \rho\omega$$

Например:

$(LFC\ C) \circ = 11 + \rho C$				$IFC\ C$			
1	0	0	0	1	0	-1	0
1	0	0	0	1	0	0	-1
0	1	0	0	0	1	-1	0
0	0	1	0	0	-1	1	0
0	0	1	0	0	0	1	-1
0	0	0	1	-1	0	0	1
0	0	1	0				
0	0	0	1				
0	0	1	0				
0	1	0	0				
0	0	0	1				
1	0	0	0				

При работе с неориентированными графами иногда пользуются представлением, определенным как операция *или* над этими плоскостями ( $\vee$ ). Это выражение эквивалентно  $|IFC\ C$ .

Матрица инцидентности  $I$  характеризуется рядом полезных свойств. Например,  $+I$  равняется нулю,  $+I$  дает разность между входными и выходными степенями каждого узла,  $\rho I$  дает число дуг, за которым следует число узлов, а  $\times/\rho I$  дает их произведение. Однако все это легко выражается через матрицу соединений, и более существенные свойства матрицы инцидентности проявляются в ее использовании в электрических сетях. Так, если дуги представляют элементы схем, включенные между узлами, и если  $V$  — это вектор напряжений в узлах, то напряжения на отдельных ветвях задаются как  $I + \times V$ , если  $BI$  — вектор токов ветвей, то вектор узловых токов задается как  $BI + \times I$ .

Обратное преобразование из матрицы инцидентности в матрицу соединений задается следующим образом:

$$CFI: D\rho(-1 + 1 \times / D) \in D \perp (1 - 1 \circ = \omega) + . \times -1 + 11 + D + \lfloor \setminus \text{Фрш}$$

Функция принадлежности множеству  $\in$  порождает булев массив той же формы, что и ее левый аргумент; этот массив показывает, какие элементы левого аргумента принадлежат правому аргументу.

### 3.5. СИМВОЛЬНАЯ ЛОГИКА

Булева функция от  $N$  аргументов может быть представлена булевым вектором из  $2 \times N$  элементов различными способами,



включая формы, иногда называемые *дизъюнктивной, конъюнктивной, эквивалентностью* и *исключающе дизъюнктивной*. Преобразование между любыми двумя из этих форм может быть представлено в сжатом виде как некоторая матрица размера  $2*N$  на  $2*N$ , сформированная из соответствующего внутреннего произведения, например  $T \vee, \wedge \emptyset T$ , где  $T \leftarrow TN$  — это «таблица истинности», построенная по функции  $T$ , описанной в А.2. Эти проблемы детально обсуждаются в [11, гл. 7].

## 4. ТОЖДЕСТВА И ДОКАЗАТЕЛЬСТВА

В этом разделе мы введем некоторые широко используемые тождества и представим формальные доказательства для части из них, в том числе для симметричных функций Ньютона и для ассоциативности внутреннего произведения, что редко доказывается формально.

### 4.1. ОТНОШЕНИЯ ДВОЙСТВЕННОСТИ ВО ВНУТРЕННЕМ ПРОИЗВЕДЕНИИ

Разработанные для сведения и развертки отношения двойственности очевидным образом обобщаются на внутренние произведения. Если функция  $DF$  двойственная к  $F$ , а  $DG$  двойственная к  $G$  по отношению к одноместной функции  $M$  с обратной функцией  $MI$  и если  $A$  и  $B$  — это матрицы, то:

$$A F. C B \leftarrow \rightarrow MI (M A) DF. DG (M B)$$

Например:

$$A \vee, \wedge B \leftrightarrow \sim (\sim A) \wedge, \vee (\sim B)$$

$$A \wedge, \vee B \leftrightarrow \sim (\sim A) \vee, \wedge (\sim B)$$

$$A \downarrow, + B \leftrightarrow - (-A) \uparrow, + (-B)$$

Двойственности для внутреннего произведения, сведения и развертки могут служить для того, чтобы избежать применения логического отрицания в различных выражениях, особенно при использовании в конъюнкции с тождествами следующего вида:

$$A \wedge (\sim B) \leftrightarrow A > B$$

$$(\sim A) \wedge B \leftrightarrow A < B$$

$$(\sim A) \wedge (\sim B) \leftrightarrow A \nabla B$$

### 4.2. ТОЖДЕСТВА РАЗБИЕНИИ

Разбиение массива ведет к ряду очевидных и полезных тождеств. Например:

$$\times/3 \ 1 \ 4 \ 2 \ 6 \leftarrow \rightarrow (\times/3 \ 1) \times (\times/4 \ 2 \ 6)$$

В более общем виде для любой ассоциативной функции  $F$

$$F/V \leftarrow \rightarrow (F/K \uparrow V) \ F \ (F/K \downarrow V)$$

$$F/V, W \leftarrow \rightarrow (F/V) \ F \ (F/W)$$

Если функция  $F$  не только ассоциативная, но и коммутативная, то нет надобности ограничивать разбиения «префиксами» и «суффиксами», и разбиение может быть построено сжатием по булеву вектору  $U$ :

$$F/V \leftarrow \rightarrow (F/U/V) \ F \ (F/(\sim U)/V)$$

Если  $E$  — пустой вектор ( $0 = \rho E$ ), то сведение  $F/E$  порождает тождественный элемент для функции  $F$ , и поэтому тождества справедливы в предельных случаях  $0 = K$  и  $0 = \vee/U$ .

Тождества разбиений очевидным образом обобщаются на матрицы. Например, если  $V$ ,  $M$  и  $A$  — массивы рангов 1, 2 и 3 соответственно, то

$$V + . \times M \leftarrow \rightarrow ((K \uparrow V) + . \times (K, 1 \downarrow \rho M) \uparrow M) + \\ + (K \downarrow V) + . \times (K, 0) \downarrow M \quad D.1$$

$$(I, J) \downarrow A + . \times V \leftarrow \rightarrow ((I, J, 0) \downarrow A) + . \times V \quad D.2$$

### 4.3. СУММИРОВАНИЕ И РАСПРЕДЕЛЕНИЕ

Рассмотрим описание и использование следующих функций:

$$\underline{N} : (\vee / < \backslash \omega \bullet \bullet = \omega) / \omega \quad D.3$$

$$\underline{S} : (\underline{N} \omega) \bullet \bullet = \omega \quad D.4$$

$$A \leftarrow 3 \ 3 \ 1 \ 4 \ 1 \\ C \leftarrow 10 \ 20 \ 30 \ 40 \ 50$$

$\underline{N} \ A$	$\underline{S} \ A$	$(\underline{S} \ A) + . \times C$
3   1   4	1   1   0   0   0	30   80   40
	0   0   1   0   1	
	0   0   0   1   0	

Функция  $\underline{N}$  выбирает из векторного аргумента его *существо*, т.е. множество различных элементов, которые в нем содержатся. Выражение  $\underline{S} A$  дает булеву «матрицу суммирования», которая связывает элементы из  $A$  с элементами существа  $A$ . Если  $A$  — вектор номеров счетов, а  $C$  — соответствующий вектор денежных сумм, то выражение  $(\underline{S} A) + . \times C$  вычисляет общие суммы, или «суммирует» сделки по нескольким номерам счетов, представленным в  $A$ .

Используемая как последующий множитель в выражении вида  $W + \cdot \times \underline{S} A$  матрица *суммирования* может служить для распределения результатов. Например, если  $F$  — функция, вычисление которой обходится дорого, и ее аргумент  $V$  содержит повторяющиеся элементы, то, может быть, более эффективно было бы применить  $F$  только к существу для  $V$  и распределить результаты тем способом, который подсказывается следующим тождеством:

$$F V \leftarrow \rightarrow (F \underline{N} V) + \cdot \times \underline{S} V \quad D.5$$

Порядок элементов из  $\underline{N}V$  такой же, как и их порядок в  $V$ , и иногда бывает удобнее использовать *упорядоченное* существо и соответствующее *упорядоченное* суммирование по формулам

$$\underline{Q}\underline{N} : \underline{N}\omega [ \Delta \omega ] \quad D.6$$

$$\underline{Q}\underline{S} : (\underline{Q}\underline{N}\omega) \circ \cdot = \omega \quad D.7$$

Тождество, соответствующее формуле D.5, имеет вид

$$F V \leftarrow \rightarrow (F \underline{O} \underline{N} V) + \cdot \times \underline{O} \underline{S} V \quad D.8$$

Функция суммирования даёт интересный результат при ее применении к функции  $I$ , описанной с помощью A.2:

$$+ / \underline{S} + \cdot \underline{T} \underline{N} \leftarrow \rightarrow (0, \iota N) ! N$$

Иначе говоря, сумма строк матрицы суммирования для сумм столбцов матрицы подмножества порядка  $N$  образует вектор биномиальных коэффициентов порядка  $N$ .

#### 4.4. ДИСТРИБУТИВНОСТЬ

Дистрибутивность одной функции относительно другой является важным математическим понятием, и теперь мы займемся вопросом представления этого понятия в общем виде. Поскольку умножение дистрибутивно относительно сложения, стоящего справа от него, то мы имеем  $a \times (b + q) \leftarrow \rightarrow ab + aq$ ; а ввиду дистрибутивности слева получаем  $(a + p) \times b \leftarrow \rightarrow ab + pb$ . Отсюда получаем более общие формулы:

$$(a + p) \times (b + q) \leftrightarrow ab + aq + pb + pq$$

$$(a + p) \times (b + q) \times (c + r) \leftrightarrow abc + abr + aqc + aqr + pbc + pbr + pqc + pqr$$

$$(a + p) \times (b + q) \times \dots \times (c + r) \leftrightarrow ab \dots c + \dots + pq \dots r$$

Воспользовавшись тем, что  $V \leftarrow A, B$  и  $W \leftarrow P, Q$  или  $V \leftarrow A, B, C$  и  $W \leftarrow P, Q, R$  и т. д., можно записать левую часть просто через сведение как  $\times/V+W$ . Для этого случая трех элементов правая часть может быть записана как сумма произведений по всем столбцам следующей матрицы:

$V[0]$	$V[0]$	$V[0]$	$V[0]$	$W[0]$	$W[0]$	$W[0]$	$W[0]$
$V[1]$	$V[1]$	$W[1]$	$W[1]$	$V[1]$	$V[1]$	$W[1]$	$W[1]$
$V[2]$	$W[2]$	$V[2]$	$W[2]$	$V[2]$	$W[2]$	$V[2]$	$W[2]$

Приведенная выше структура распределения  $V$  и  $W$  в точности совпадает с распределением нулей и единиц в матрице  $T \leftarrow T \rho V$ , и поэтому произведения вдоль столбцов задаются как  $(V \times . * \sim T) \times (W \times . * T)$ . Следовательно,

$$\times/V+W \leftarrow \rightarrow +/(V \times . * \sim T) \times W \times . * T \leftarrow T \rho V \quad D.9$$

Теперь мы представим формально индуктивное доказательство для D.9, приняв индуктивное предположение, что D.9 справедливо для всех  $V$  и  $W$  формы  $N$  (т. е.  $\bigwedge/N = (\rho V), (\rho W)$ ), и доказав, что оно справедливо для формы  $N+1$ , т. е. для  $X, V$  и  $Y, W$ , где  $X$  и  $Y$  — это произвольные скаляры.

Для использования в индуктивном доказательстве мы сначала дадим рекурсивное описание функции  $T$ , эквивалентное A.2 и основанное на следующем утверждении; если  $M \leftarrow T 2$  является результатом порядка 2, то

$M$							
0	0	1	1				
0	1	0	1				
$0, [1]M$				$1, [1]M$			
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1
$(0, [1]M), (1, (1)M)$							
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1

Итак:

$$\mathcal{T}:(0, [1]T), (1, [1]T + \mathcal{T}\omega - 1): 0 = \omega: 0 \quad 1\rho 0$$

D.10

$$\begin{aligned} &+ / ((C+X, V) \times . \sim Q) \times D \times . \sim Q + \mathcal{T}\rho (D+Y, W) \\ &+ / (C \times . \sim Z, U) \times D \times . (Z+0, [1] T), U+1, [1] T + \mathcal{T}\rho W \\ &+ / ((C \times . \sim Z), C \times . \sim U) \times (D \times . \sim Z), D \times . \sim U \\ &+ / ((C \times . \sim Z), C \times . \sim U) \times ((Y \times 0) \times W \times . \sim T), (Y \times 1) \times W \times . \sim T \\ &+ / ((C \times . \sim Z), C \times . \sim U) \times (W \times . \sim T), Y \times W \times . \sim T \\ &+ / ((X \times V \times . \sim T), V \times . \sim T) \times (W \times . \sim T), Y \times W \times . \sim T \\ &+ / (X \times (V \times . \sim T) \times W \times . \sim T), (Y \times (V \times . \sim T) \times W \times . \sim T) \\ &+ / (X \times \sim / V + W), (Y \times \sim / V + W) \\ &+ / (X, Y) \times \sim / V + W \\ &\times / (X+Y), (V+W) \\ &\times / (X, V) + (Y, W) \end{aligned}$$

D.10

Note 1

Note 2

Note 2

Note 3

Induction hypothesis

$(X \times S), (Y \times S) \leftrightarrow (X, Y) \times S$

Definition of  $\times /$

+ distributes over ,

Note 1:  $M + . \times N, P \leftarrow \rightarrow (M + . \times N), M + . \times P$  (тождество разбиения для матриц)

Note 2:  $V + . \times M \leftarrow \rightarrow ((1 \uparrow V) + . \times (1, 1 \downarrow \rho M) \uparrow M) + (1 \downarrow V) + . \times 1 \downarrow M$  (тождество разбиения для матриц и описание  $C, D, Z$  и  $U$ )

Note 3:  $(V; W) \times P, Q \leftarrow \rightarrow (V \times P), W \times Q$

Чтобы завершить индуктивное доказательство, нам нужно показать, что предполагаемое тождество D.9 справедливо для некоторого значения  $N$ . Если  $N=0$ , то векторы  $A$  и  $B$  пустые и поэтому  $X, A \leftarrow \leftarrow, X$  и  $Y, B \leftarrow \rightarrow, Y$ . Следовательно, левая часть приобретает вид  $+ / X + Y$  или просто  $X + Y$ . Правая часть приобретает вид  $+ / (X \times . \sim Q) \times Y \times . Q$ , где  $\sim Q$  — это однострочная матрица  $1 \ 0$ , а  $Q$  — это  $0 \ 1$ . Поэтому правая сторона эквивалентна  $+ / (X, 1) \times (1, Y)$  или  $X + Y$ . Может оказаться поучительным аналогичное исследование случая  $N=1$ .

#### 4.5. СИММЕТРИЧНЫЕ ФУНКЦИИ НЬЮТОНА

Если  $X$  — это скаляр, а  $R$  — любой вектор, то  $\times / X - R$  является полиномом над  $X$  с корнями  $R$ . Поэтому он эквивалентен некоторому полиному  $\underline{C} \underline{P} X$  и признание эквивалентности означает, что  $\underline{C}$  представляет собой функцию от  $R$ . Теперь мы используем D.8 и D.9 для вывода этой функции, которая обычно строится на основе симметричных функций Ньютона:

$$\begin{aligned} &\times / X - R \\ &\times / X + (-R) \\ &+ / (X \times . \sim T) \times (-R) \times . \sim T + \mathcal{T} \rho R \\ &(X \times . \sim T) + . \times P + (-R) \times . \sim T \\ &(X \times S + \sim / \sim T) + . \times P \\ &((X \times \underline{Q} \underline{S}) + . \times \underline{Q} \underline{S} S) + . \times P \\ &(X \times \underline{Q} \underline{S} S) + . \times ((\underline{Q} \underline{S} S) + . \times P) \\ &(X \times 0, 1 \rho R) + . \times ((\underline{Q} \underline{S} S) + . \times P) \\ &((\underline{Q} \underline{S} S) + . \times P) \underline{P} X \\ &((\underline{Q} \underline{S} \sim / \sim T) + . \times ((-R) \times . \sim T + \mathcal{T} \rho R)) \underline{P} X \end{aligned}$$

D.9

Def of  $+ . \times$

Note 1

D.8

$+ . \times$  is associative

Note 2

B.1 (polynomial)

Def of  $S$

and  $P$

Note 1: Если  $X$  — это скаляр и  $B$  — булев вектор, то  $X \times . * B \leftarrow \rightarrow X * \times / B$ .

Note 2: Поскольку  $T$  — булева матрица и содержит  $\rho R$  строк, то суммы ее столбцов ранжируются от 0 до  $\rho R$ , и поэтому их упорядоченная сущность равна 0,  $\rho R$ .

#### 4.6. ДВУМЕСТНАЯ ТРАНСПОЗИЦИЯ

Обозначаемая через  $\otimes$  двуместная транспозиция представляет собой обобщение одноместной транспозиции, которое представляет оси правого аргумента и (или) формирует «секторы» правого аргумента слиянием определенных осей, определяемых левым аргументом. Мы вводим здесь эту операцию как удобное средство изучения свойств внутреннего произведения.

Двуместная транспозиция будет описана формально с помощью функции выбора

$$SF: (, \omega) [1 + (\rho\omega) \perp \alpha - 1]$$

извлекающей из своего правого аргумента элемент, индексы которого задаются векторным левым аргументом. Разумеется, его форма должна равняться рангу правого аргумента. Ранг результата  $K \otimes A$  равен  $\Gamma / K$ , и если  $I$  — это любой подходящий левый аргумент функции выбора  $I SF I \otimes A$ , то:

$$I SF K \otimes A \leftarrow \rightarrow (I[K]) SF A \quad D.11$$

Например, если  $M$  — это матрица, то  $2 \ 1 \otimes M \leftarrow \rightarrow \otimes M$  и  $1 \ 1 \otimes M$  является диагональю матрицы  $M$ ; если  $T$  — массив ранга 3, то  $1 \ 2 \ 2 \otimes T$  — это матрица «диагонального разреза»  $T$ , полученная выделением последних двух осей, а вектор  $1 \ 1 \ 1 \otimes T$  является главной диагональю для  $T$ .

В дальнейшем будет использовано следующее тождество:

$$J \otimes K \otimes A \leftarrow \rightarrow (J[K]) \otimes A \quad D.12$$

Доказательство:

$$I SF J \otimes K \otimes A$$

$$(I[J]) SF K \otimes A$$

$$((I[J])[K]) SF A$$

$$(I[(J[K])]) SF A$$

$$I SF (J[K]) \otimes A$$

Definition of  $\otimes$  (D.11).

Definition of  $\otimes$ .

Indexing is associative

Definition of  $\otimes$ .

#### 4.7. ВНУТРЕННИЕ ПРОИЗВЕДЕНИЯ

Следующие доказательства сформулированы только для матричных аргументов и для конкретного внутреннего произведения  $+ \cdot \times$ . Они легко обобщаются на массивы более высокого

ранга и на другие внутренние произведения  $F.G$ , где  $F$  и  $G$  должны обладать только свойствами, предполагаемыми в доказательствах для  $+$  и  $\times$ .

Следующее тождество (известное в математике как сумма по матрицам, сформированным как (внешние) произведения столбцов из первого аргумента на соответствующие строки из второго аргумента) будет использоваться при установлении ассоциативности и дистрибутивности внутреннего произведения:

$$M+. \times N \leftarrow \rightarrow +/1\ 3\ 3\ 2 \mathbb{Q} M\circ. \times N \quad D.13$$

Доказательство:  $(I, J)SF M+. \times N$  описывается как сумма по  $V$ , где  $V[K] \leftarrow \rightarrow M[I, K] \times N[K, J]$ . Аналогично

$$(I, J)SF +/1\ 3\ 3\ 2 \mathbb{Q} M\circ. \times N$$

является суммой по вектору  $W$ , такому, что

$$W[K] \leftarrow \rightarrow (I, J, K)SF\ 1\ 3\ 3\ 2 \mathbb{Q} M\circ. \times N$$

Итак:

$$\begin{aligned} & W[K] \\ & (I, J, K)SF\ 1\ 3\ 3\ 2 \mathbb{Q} M\circ. \times N \\ & (I, J, K)[1\ 3\ 3\ 2]SF\ M\circ. \times N \\ & (I, K, K, J)SF\ M\circ. \times N \\ & M[I; K] \times N[K; J] \\ & V[K] \end{aligned}$$

Матричное произведение дистрибутивно относительно сложения следующим образом:

$$M+. \times (N+P) \leftarrow \rightarrow (M+. \times N) + (M+. \times P) \quad D.14$$

Доказательство:

$$\begin{aligned} & M+. \times (N+P) \\ & +/(J+ 1\ 3\ 3\ 2) \mathbb{Q} M\circ. \times N+P \quad D.13 \\ & +/J \mathbb{Q} (M\circ. \times N) + (M\circ. \times P) \quad \times \text{ distributes over } + \\ & +/(J \mathbb{Q} M\circ. \times N) + (J \mathbb{Q} M\circ. \times P) \quad \mathbb{Q} \text{ distributes over } + \\ & (+/J \mathbb{Q} M\circ. \times N) + (+/J \mathbb{Q} M\circ. \times P) \quad + \text{ is assoc and comm} \\ & (M+. \times N) + (M+. \times P) \quad D.13 \end{aligned}$$

Матричное произведение ассоциативно согласно формуле

$$M+. \times (N+. \times P) \leftarrow \rightarrow (M+. \times N) +. \times P \quad D.15$$

Доказательство: Сначала сводим каждую строку к суммам по разделам внешнего произведения и затем сравниваем суммы.  
Комментирование второго сведения предоставляется читателям:

$$\begin{aligned}
 &M+. \times (N+. \times P) \\
 &M+. \times +/1 \ 3 \ 3 \ 2Q N \circ. \times P && \text{D.12} \\
 &+/1 \ 3 \ 3 \ 2Q M \circ. \times +/1 \ 3 \ 3 \ 2Q N \circ. \times P && \text{D.12} \\
 &+/1 \ 3 \ 3 \ 2Q +/M \circ. \times 1 \ 3 \ 3 \ 2Q N \circ. \times P && \times \text{ distributes over } + \\
 &+/1 \ 3 \ 3 \ 2Q +/1 \ 2 \ 3 \ 5 \ 5 \ 4Q M \circ. \times N \circ. \times P && \text{Note 1} \\
 &+ / + / 1 \ 3 \ 3 \ 2 \ 4 \ Q 1 \ 2 \ 3 \ 5 \ 5 \ 4Q M \circ. \times N \circ. \times P && \text{Note 2} \\
 &+ / + / 1 \ 3 \ 3 \ 4 \ 4 \ 2Q M \circ. \times N \circ. \times P && \text{D.12} \\
 &+ / + / 1 \ 3 \ 3 \ 4 \ 4 \ 2Q (M \circ. \times N) \circ. \times P && \times \text{ is associative} \\
 &+ / + / 1 \ 4 \ 4 \ 3 \ 3 \ 2Q (M \circ. \times N) \circ. \times P && + \text{ is associative and commutative}
 \end{aligned}$$

$$\begin{aligned}
 &(M+. \times N) +. \times P \\
 &(+ / 1 \ 3 \ 3 \ 2Q M \circ. \times N) +. \times P \\
 &+/1 \ 3 \ 3 \ 2Q (+/1 \ 3 \ 3 \ 2Q M \circ. \times N) \circ. \times P \\
 &+/1 \ 3 \ 3 \ 2Q +/1 \ 5 \ 5 \ 2 \ 3 \ 4Q (M \circ. \times N) \circ. \times P \\
 &+ / + / 1 \ 3 \ 3 \ 2 \ 4Q 1 \ 5 \ 5 \ 2 \ 3 \ 4Q (M \circ. \times N) \circ. \times P \\
 &+ / + / 1 \ 4 \ 4 \ 3 \ 3 \ 2Q (M \circ. \times N) \circ. \times P
 \end{aligned}$$

$$\text{Note 1: } +/M \circ. \times JQ A \leftrightarrow +/((1\rho\rho M), J + \rho\rho M)Q M \circ. \times A$$

$$\text{Note 2: } JQ +/A \leftrightarrow +/(J, 1 + \lceil J)Q A$$

#### 4.8. ПРОИЗВЕДЕНИЕ ПОЛИНОМОВ

Тождество В.2, использованное для умножения полиномов, будет теперь разработано формально

$$\begin{aligned}
 &(B \ P \ X) \times (C \ P \ X) \\
 &(+/B \times X \star E +^{-1} 1 + \rho B) \times (+/C \times X \star F +^{-1} 1 + \rho C) && \text{B.1} \\
 &+ / + / (B \times X \star E) \circ. \times (C \times X \star F) && \text{Note 1} \\
 &+ / + / (B \circ. \times C) \times ((X \star E) \circ. \times (X \star F)) && \text{Note 2} \\
 &+ / + / (B \circ. \times C) \times (X \star (E \circ. + F)) && \text{Note 3}
 \end{aligned}$$

Примечание 1:  $(+/V) \times (+/W) \leftrightarrow + / + / V \circ. \times X$ , потому что  $\times$  распределяется по  $+$  и операция  $+$  ассоциативна и коммутативна, или см. доказательство в [12, с. 21].

Примечание 2: Эквивалентность  $(P \times V) \circ. \times (Q \times W)$  и  $(P \circ. \times Q) \times (V \circ. \times W)$  может быть установлена исследованием типового элемента из каждого произведения.

Примечание 3:  $(X \star I) \times (X \star J) \leftrightarrow X \star (I + J)$ .

Приведенное выше доказательство представлено в сокращенной форме Ортом [13, р. 52], который описал также функции для композиции полиномов.

#### 4.9. ПРОИЗВОДНАЯ ОТ ПОЛИНОМА

Полиномиальные функции удобно использовать в вводном курсе численного анализа, потому что они пригодны для аппрок-



симации множества полезных функций и потому что класс этих функций замкнут относительно сложения, умножения, композиции, дифференцирования и интегрирования. Однако в курсе элементарных вычислений рассмотрение таких функций обычно запаздывает, так как подход к производной полинома осуществляется, как отмечалось в разд. 2, окольным путем, т. е. через последовательность более общих результатов.

Ниже показан вывод производной полинома непосредственно из выражения для тангенса угла наклона секущей линии через точки  $X$ ,  $F X$  и  $(X+Y)$ ,  $F(X+Y)$ :

$$\begin{aligned}
 & ((C \text{ } P \text{ } X+Y) - (C \text{ } P \text{ } X)) + Y \\
 & ((C \text{ } P \text{ } X+Y) - (C \text{ } P \text{ } X+0)) + Y \\
 & ((C \text{ } P \text{ } X+Y) - ((0 * J) + . * (A + DS \text{ } J \text{ } . ! J + ^{-1} + \text{ } \rho C) + . * C) \text{ } P \text{ } X) + Y \quad \text{B.6} \\
 & (((Y * J) + . * M) \text{ } P \text{ } X) - ((0 * J) + . * M + A + . * C) \text{ } P \text{ } X) + Y \quad \text{B.6} \\
 & (((Y * J) + . * M) - (0 * J) + . * M) \text{ } P \text{ } X) + Y \quad \text{P dist over -} \\
 & (((Y * J) - 0 * J) + . * M) \text{ } P \text{ } X) + Y \quad + . * \text{ dist over -} \\
 & ((0, Y * 1 + J) + . * M) \text{ } P \text{ } X) + Y \quad \text{Note 1} \\
 & (((Y * 1 + J) + . * 1 \text{ } 0 \text{ } 0 + M) \text{ } P \text{ } X) + Y \quad \text{D.1} \\
 & (((Y * 1 + J) + . * (1 \text{ } 0 \text{ } 0 + A) + . * C) \text{ } P \text{ } X) + Y \quad \text{D.2} \\
 & ((Y * 1 + J - 1) + . * (1 \text{ } 0 \text{ } 0 + A) + . * C) \text{ } P \text{ } X \quad (Y * A) + Y \leftrightarrow Y * A - 1 \\
 & ((Y * ^{-1} + ^{-1} + \rho C) + . * (1 \text{ } 0 \text{ } 0 + A) + . * C) \text{ } P \text{ } X \quad \text{Def of } J \\
 & (((Y * ^{-1} + ^{-1} + \rho C) + . * 1 \text{ } 0 \text{ } 0 + A) + . * C) \text{ } P \text{ } X \quad \text{D.15}
 \end{aligned}$$

Note 1:  $0 * 0 \leftrightarrow 1 \leftrightarrow Y * 0$  and  $\wedge / 0 = 0 * 1 + J$

Производная представляет собой предельное значение тангенса угла наклона секущей при  $Y \rightarrow 0$ , и последнее из вышеприведенных выражений поддается в этом случае вычислению, так как если  $E \leftarrow -1 + ^{-1} + \rho C$  есть вектор экспонент для  $Y$ , то все элементы из  $E$  неотрицательные. Кроме того,  $0 * E$  сводится к единице, за которой следуют нули, а поэтому внутреннее произведение для  $1 \text{ } 0 \text{ } 0 \downarrow A$  сводится к первой плоскости из  $1 \text{ } 0 \text{ } 0 \downarrow A$  или, что эквивалентно, ко второй плоскости из  $A$ .

Если  $B \leftarrow J \text{ } . ! J \leftarrow -1 + \rho C$  является матрицей биномиальных коэффициентов, то  $A$  — это  $DS \text{ } B$  и, согласно описанию  $DS$  в В.5, вторая плоскость из  $A$  представляет собой  $B \times 1 = -J \text{ } . - J$ , т. е. матрицу  $B$ , в которой все элементы, за исключением первой супердиагонали, заменены нулями. Поэтому окончательное выражение для коэффициентов полинома, который является производной от полинома  $CP_{\omega}$ , имеет вид

$$((J \text{ } . ! J) \times 1 = -J \text{ } . - J + ^{-1} + \rho C) + . * C$$

Например:

$$\begin{array}{r}
 C + 5 \quad 7 \quad 11 \quad 13 \\
 (J \bullet \dots J) \times 1 = -J \bullet \dots -J +^{-1} 1 + \rho C \\
 0 \quad 1 \quad 0 \quad 0 \\
 0 \quad 0 \quad 2 \quad 0 \\
 0 \quad 0 \quad 0 \quad 3 \\
 0 \quad 0 \quad 0 \quad 0 \\
 ((J \bullet \dots J) \times 1 = -J \bullet \dots -J +^{-1} 1 + \rho C) + \dots \times C \\
 7 \quad 22 \quad 39 \quad 0
 \end{array}$$

Поскольку супердиагональ матрицы биномиальных коэффициентов  $(iN) \circ iN$  представляет собой  $(-1 + iN - 1) iN - 1$ , или просто  $iN - 1$ , окончательный результат равен  $1 \text{ ф} C \times -1 + \text{ф} C$  в соответствии с приведенным выше выводом.

Завершая обсуждение доказательств, снова подчеркнем тот факт, что все утверждения (выражения) в предшествующих доказательствах исполнимы (вычислимы), и поэтому для обнаружения ошибок можно использовать компьютер. Например, воспользовавшись узловым описанием канонической функции [4, с. 81], можно следующим образом описать некую функцию  $F$ , шагами вывода которого являются четыре строки из предыдущего доказательства:

$$\begin{aligned} & \nabla F \\ [1] & ((C \text{ E } X \text{ r } Y) - (C \text{ E } X)) \div Y \\ [2] & ((C \text{ E } X + Y) - (C \text{ E } X + 0)) \div Y \\ [3] & ((C \text{ E } X + Y) - ((0 \div J) + . \times (A \div DS \text{ J} \bullet . ! J \bullet^{-1} + 1 \text{ r } C) + . \times C) \text{ E } X) \div Y \\ [4] & (((Y \div J) + . \times M) \text{ E } X) - ((0 \div J) + . \times M + A + . \times C) \text{ E } X) \div Y \\ & \nabla \end{aligned}$$

Затем можно выполнить эти шаги доказательства, присвоив значения переменным и вычислив  $F$  следующим образом:

	C+5	2	3	1							
	Y+5										
	X+3				X+110						
	F				F						
132		66	96	132	174	222	276	336	402	474	552
132		66	96	132	174	222	276	336	402	474	552
132		66	96	132	174	222	276	336	402	474	552
132		66	96	132	174	222	276	336	402	474	552

Между строками могут быть **включены** комментарии, не влияющие на выполнение.

## 5. ЗАКЛЮЧЕНИЕ

В предшествующих разделах была предпринята попытка разработать тезис о том, что свойства исполнимости и универсаль-

ности, присущие языкам программирования, можно объединить в одном языке с хорошо известными свойствами математической нотации, которые делают ее столь эффективным инструментом мышления. Этому важному вопросу следует уделять дальнейшее внимание вне зависимости от успеха или неудачи данной попытки разработать его в терминах языка APL.

В частности, я хотел бы надеяться, что другие исследователи станут заниматься тем же вопросом с помощью иных языков программирования и обычной математической нотации. Если эти рассмотрения окажутся нацеленными на те общие аспекты, которыми мы занимались здесь, то можно будет провести некоторые объективные сравнения языков. Уже доступны для сравнения альтернативные рассмотрения некоторых представленных здесь тем. Например, Кернер [7] выразил алгоритм C.3 как на Алголе, так и в общепринятой математической нотации.

Этот заключительный раздел является более общим и посвящен сравнениям с математической нотацией, проблемам введения нотации, обобщениям языка APL, которые способствовали бы дальнейшему увеличению его полезности, и обсуждению вариантов представления материала предшествующих разделов.

## 5.1. СРАВНЕНИЕ С ОБЫЧНОЙ МАТЕМАТИЧЕСКОЙ НОТАЦИЕЙ

Для любого недостатка, замеченного в математической нотации, вероятно, можно найти пример его исправления в некотором конкретном разделе математики или в некоторой конкретной публикации; проводимые здесь сравнения ориентированы на более общие и типичные применения математической нотации.

Язык APL подобен обычной математической нотации во многих важных отношениях: в нем используются функции с явными аргументами и явными результатами, употребляются выражения, в которых участвуют функции от результатов других функций; для часто используемых функций имеются графические символы, применяются операторы, которые, подобно математическим операторам дифференцирования и свертки, применяются к функциям для получения функций.

Особенность подхода к функциям в APL состоит в обеспечении точного формального механизма для описания новых функций. Используемая в этой статье форма прямого определения, вероятно, лучше всего подходит для целей демонстрации и анализа, но каноническая форма, упоминаемая во введении и описанная в [4, с. 81], часто оказывается более удобной для других целей.

Интерпретация сложных выражений в APL согласуется

с обычной нотацией в употреблении скобок, но отличается отсутствием иерархии, обеспечивающим единообразный подход ко всем функциям (как определенным пользователем, так и примитивным), и применением одного и того же правила к одноместным и двуместным функциям: правым аргументом функции является все выражение, стоящее справа от нее. Из этого правила вытекает важное следствие, состоящее в том, что любая часть выражения, свободная от скобок, может *аналитически* читаться слева направо (потому что на любом этапе самая левая функция является «внешней», или общей, функцией, которая должна быть применена к результату, стоящему справа от нее) и *конструктивно* читаться справа налево (поскольку легко видеть, что это правило эквивалентно правилу *исполнения* справа налево).

Хотя Кайори [2] в своей двухтомной истории математических нотаций даже не упомянул правила для порядка исполнения, тем не менее представляется разумным предположить, что мотивация привычной иерархии (возведение в степень выполняется перед умножением, а умножение — перед сложением или вычитанием) возникла из желания записывать полиномы без скобок. Удобство применения векторов при выражении полиномов, например  $+ / C \times X * E$ , во многом способствует устранению этой мотивации. Кроме того, принятое в языке APL правило также обосновывает применимость без скобок эффективной схемы Горнера для выражения полинома:

$$+ / 3 \ 4 \ 2 \ 5 \times X * 0 \ 1 \ 2 \ 3 \leftarrow \rightarrow 3 + X \times 4 + X \times 2 + X \times 5.$$

В обеспечении графических символов для общеупотребимых функций APL идет значительно дальше и предусматривает такие символы для функций (например, для степенной функции), которые неявно отвергаются в математике. Это обстоятельство становится важным при введении операторов; в предшествующих разделах внутреннее произведение  $\times *$  (в котором должен применяться символ для степени) играет такую же роль, как и обычное внутреннее произведение  $+. \times$ . Запрет на отбрасывание символов функций (например, символа  $\times$ ) делает возможным недвусмысленное употребление многосимвольных имен для переменных и функций.

В отношении использования массивов язык APL сходен с математической нотацией, но характеризуется большей систематичностью. Например,  $V + W$  имеет одинаковый смысл в обеих нотациях, и в APL определения других функций обобщаются аналогичным поэлементным способом. Однако в математике такие выражения, как  $V \times W$  и  $V * W$ , определяются по-другому или вовсе не определены.

$$\sum_{j=1}^n j \cdot 2^{-j}$$

$$1 \cdot 2 \cdot 3 + 2 \cdot 3 \cdot 4 + \dots n \text{ членов} \leftarrow \rightarrow \frac{1}{4} n(n+1)(n+2)(n+3)$$

$$1 \cdot 2 \cdot 3 \cdot 4 + 2 \cdot 3 \cdot 4 \cdot 5 + \dots n \text{ членов} \leftarrow \rightarrow \frac{1}{5} n(n+1)(n+2)(n+3)(n+4)$$

$$\frac{\left[ \frac{x-a}{N} \right]^{-j}}{\Gamma(-q)} \sum_{j=0}^{N-1} \frac{\Gamma(j-q)}{\Gamma(j+1)} f \left( x - j \left[ \frac{x-a}{N} \right] \right)$$

Рис. 3.

Так,  $Y \times W$  обычно означает *векторное произведение* [14]. На языке APL оно может быть выражено различными способами. Определение

$$VP: ((1\phi\alpha) \times \neg 1\phi\omega) \neg (\neg 1\phi\alpha) \times 1\phi\omega$$

обеспечивает удобную основу для очевидного доказательства того, что  $VP$  является «антикоммутативным» (т. е.  $V VP W \leftarrow \rightarrow \neg W VP V$ ) и (с учетом того, что  $\neg 1\phi X \leftarrow \rightarrow 2\phi X$  для трехэлементных векторов) позволяет легко доказать, что в трехмерном пространстве  $V$  и  $W$  ортогональны своему векторному произведению, т. е.  $\wedge/0 = V +. \times V VP W$  и  $\wedge/0 = W +. \times V VP W$ .

Язык APL более систематичен также в использовании операций для получения функций на массивах; сведение обеспечивает эквиваленты для обозначений sigma и pi (в виде  $+ /$  и  $\times /$ ) и для множества аналогичных полезных ситуаций; внешнее произведение обобщает внешнее произведение из тензорного анализа на функции, отличающиеся от  $\times$ , а внутреннее произведение обобщает матричное произведение ( $+ . \times$ ) на многие случаи, например на  $\vee . \wedge$  и  $\perp . +$ , для которых часто вводятся описания применительно к конкретному случаю.

Сходство между языком APL и традиционной нотацией становится яснее, если выучить несколько довольно механических подстановок; поучительно и преобразование математических выражений. Например, в таком выражении, как показанное первым на рис. 3, производится простая подстановка  $\iota N$  вместо всякого появления  $j$  и замена знака  $\Sigma$  на  $+ /$ . Итак:

$$+ / (\iota N) \times 2 * \neg \iota N \quad \text{или} \quad + / J \times 2 * \neg J \leftarrow \iota N$$

В руководстве по суммированию рядов [15] представлены интересные выражения для таких упражнений, особенно если результаты можно вычислить на компьютере. Например, на с. 8

и 9 из [15] приведены тождества, показанные во втором и третьем примерах на рис. 3. Их можно было бы записать так:

$$+/\times/(-1+\iota N)\circ.\iota 3\longleftrightarrow(\times/N+0,\iota 3)\div 4$$

$$+/\times/(-1+\iota N)\circ.\iota 4\longleftrightarrow(\times/N+0,\iota 4)\div 5$$

В совокупности они подсказывают следующее тождество:

$$+/\times/(-1+\iota N)\circ.\iota K\longleftrightarrow(\times/N+0,\iota K)\div K+1$$

Читатель может попытаться переформулировать это общее тождество (или хотя бы его частный случай при  $K=0$ ) в нотации из [15].

Последнее выражение на рис. 3 заимствовано из исследования по функциональному исчислению [16] и представляет аппроксимацию для производной порядка  $q$  от функции  $f$ . Оно может быть записано так:

$$\{S^*-Q\}\times +/(J!J-1+Q)\times F X-(J\leftarrow^{-1}+\iota N)\times S\leftarrow(X-A)\div N$$

Преобразование на языке APL сводится к простому использованию  $\iota N$  (как предлагалось выше) в сочетании с очевидным тождеством, сводящим несколько появлений гамма-функции в одно применение функции биномиальных коэффициентов «!», область определения которой, разумеется, не ограничивается целыми числами.

В предыдущем выражении положительное значение параметра  $Q$  указывает порядок производной, а отрицательное — порядок интеграла (от  $A$  до  $X$ ). Дробные значения задают дробные производные и интегралы, и, описав сначала функцию  $F$  и присвоив подходящие значения для  $N$  и  $A$ , можно использовать для вычислительного эксперимента с рассмотренными в [16] производными следующую функцию:

$$OS:(S^*-\alpha)\times +/(J!J-1+\alpha)\times F\omega-(J\leftarrow^{-1}+\iota N)\times \\ \times S\leftarrow(\omega-A)\div N$$

Несмотря на широкое использование «формальных» манипуляций в математической нотации, по-настоящему формальные манипуляции посредством явных алгоритмов весьма затруднительны. В этом отношении язык APL оказывается гораздо более подходящим. Например, в разд. 2 мы видели, что производная от полиномиального выражения  $(\omega\circ.*^{-1}+\iota\alpha)+.\times\alpha$  задается как  $(\omega\circ.*^{-1}+\iota\alpha)+.\times 1\phi\alpha\times^{-1}+\iota\alpha$ , и набор функций для формального дифференцирования выражений на APL, представленный в [13], занимает менее одной страницы. Другие примеры функций для формальных манипуляций встречаются в [17] применительно к операторам моделирования для векторного исчисления.

Дальнейшее рассмотрение связи с математической нотацией можно найти в [3] и в статье «Алгебра как язык» [6, с. 325].

Заключительное замечание касается печати, которая является серьезной проблемой для традиционной нотации. Хотя в языке APL применяются некоторые символы, как правило еще не доступные для издателей, в нем задействованы только 88 основных символов плюс несколько составных символов, сформированных суперпозициями пар основных символов. Кроме того, он не предъявляет таких требований, как подстрочные и надстрочные линии и уменьшенные шрифты для нижних и верхних индексов.

## 5.2. ВВЕДЕНИЕ НОТАЦИИ

В самом начале этой статьи предлагалось оценивать удобство нотации по легкости ее включения в контекст, и мы просили читателя наблюдать процесс введения нотации APL. Полезность такого критерия вполне можно счесть тривиальным фактом, но тем не менее она требует некоторого пояснения.

Во-первых, целевая нотация, предусмотренная именно для функций, нужных в некоем конкретном деле, тоже легко вводилась бы в контекст. Необходимо задать дополнительные вопросы, относящиеся к общему объему требуемой нотации, глубине структуры нотации и к тому, в какой мере нотация, введенная для специфической цели, оказывается полезной в более общих случаях.

Во-вторых, важно различать трудность описания и изучения какого-то подмножества обозначений и трудность овладения следствиями. Например, легко изучить правила вычисления произведения матриц, но совсем другое и притом гораздо более трудное дело — овладеть его следствиями (такими, как ассоциативность, дистрибутивность относительно сложения и возможность представлять линейные функции и геометрические операции).

В самом деле, сама содержательность нотации может затруднить ее изучение из-за обилия свойств, исследование которых она подсказывает. Например, нотация  $+\times$  для произведения матриц не может затруднить освоение правил его вычисления, потому что эта нотация по крайней мере напоминает, что процесс сводится к суммированию произведений, однако любое обсуждение свойств произведения матриц в терминах этой нотации не может не поставить множество вопросов, таких, например, как: ассоциативна ли операция  $\vee \wedge$ ? относительно какой операции она дистрибутивна? верно ли тождество  $B \vee \wedge C \leftarrow \rightarrow \emptyset (\emptyset C) \vee \wedge \emptyset B$ ?

### 5.3. ОБОБЩЕНИЯ ДЛЯ APL

Чтобы гарантировать корректность и широкую доступность на существующих вычислительных машинах нотации, используемой в этой статье, мы ограничились современным состоянием языка APL, описанным в [4], и более формальным стандартом, опубликованным STAPL, техническим комитетом ACM SIGPLAN по языку APL [17]. Здесь мы кратко прокомментируем потенциальные возможности, которые увеличили бы удобство этой нотации для рассматриваемых нами аспектов и в большей степени приспособили бы ее для других приложений, например для обычного и векторного исчисления.

Один тип расширения уже предлагался, когда мы продемонстрировали выполнение примера (корни полинома) в системе APL, основанной на комплексных числах. Это расширение не влечет за собой изменений функциональных символов, хотя и потребуется расширить области определения некоторых функций. Например,  $|X$  будет давать модули не только вещественных, но и комплексных аргументов,  $+X$  даст сопряженное значение для комплексного аргумента наряду с тем тривиальным результатом, который ныне получается для вещественных аргументов; соответствующим образом будут обобщены и элементарные функции, как видно из применения  $*$  в приведенном примере. Подразумевается возможность осмысленного включения примитивных функций для нулей полиномов, а также для собственных значений и собственных векторов матриц.

Другой тип, также предложенный в предшествующих разделах, включает функции, определенные для конкретных целей, которые могли бы пригодиться для общего пользования. Примерами являются функция  $N$  *сущность*, определенная в D.3, и функция  $S$  *суммирования*, определенная в D.4. Эти и другие обобщения обсуждаются в [18]. Макдоннелл [19] предложил обобщения функций *и* и *или* на небулевы значения, чтобы  $A \vee B$  являлось наибольшим общим делителем (GCD) для  $A$  и  $B$ , а  $A \wedge B$  являлось наименьшим общим кратным (LCM). Функции GCD и LCM, описанные в разд. 3, могли бы быть потом описаны просто как  $GCD: \vee/\omega$  и  $LCM: \wedge/\omega$ .

Более общая линия развития затрагивает операции; она проиллюстрирована в предыдущем разделе на примерах операций сведения, внутреннего произведения и внешнего произведения. Обсуждение операций, ныне включенных в APL, можно найти в [20] и [17]; предлагаемые новые операции для векторного исчисления рассматриваются в [17], а другие обсуждаются в [18] и [17].



#### 5.4. СПОСОБ ПРЕДСТАВЛЕНИЯ

В предшествовавших разделах рассмотрен ряд несложных тем, причем упор делался на ясность, а не на эффективность получаемых в результате алгоритмов. Оба этих качества заслуживают дополнительных пояснений.

Несколько иная и, возможно, более реалистичная проверка нотации связана с рассмотрением более сложных тем в объеме, достаточном, скажем, для одно- или двухсеместрового курса. В частности, это позволяет лучше оценить, какой объем нотации можно ввести в рамках обычного учебного курса.

Такие анализы проведены для ряда тем, включая высшую алгебру [6], элементарный анализ [5], функциональный анализ [13], проектирование цифровых систем [21], резистивные схемы [10] и кристаллографию [22]. Во всех этих случаях отмечается легкость введения нужной нотации, а в одном обсуждается опыт ее использования. Профессор Блейв, обсуждая проектирование цифровых систем [21], утверждает, что «APL позволяет описывать, что на самом деле происходит в сложной системе... Язык APL особенно удобен для этой цели, так как допускает выражения, относящиеся к верхнему уровню архитектуры, к нижнему уровню и ко всем промежуточным уровням... Изучение языка окупается как внутри, так и вне области проектирования компьютеров».

Пользователи компьютеров и языков программирования часто в первую очередь обращают внимание на эффективность исполнения алгоритмов и поэтому могли бы в итоге пренебречь многими представленными здесь алгоритмами. Такое пренебрежение оказалось бы недальновидным, потому что ясная формулировка алгоритма обычно может служить основой, из которой легко выводятся более эффективные алгоритмы. Например, для функции STEP из разд. 3.2 можно значительно повысить эффективность, выполнив подстановки вида  $B \left[ \frac{\cdot}{\cdot} \right] M$  для  $(\left[ \frac{\cdot}{\cdot} \right] M) + \times B$ , а в выражениях, содержащих  $+ / C \times X * - 1 + + \uparrow C$ , можно подставить  $X \perp \otimes C$  или, если принят противоположный порядок коэффициентов, выражение  $X \perp C$ .

Можно производить и более сложные преобразования. Метод Кернера (С.3) получается из довольно очевидного, хотя формально и не установленного тождества. Аналогично использование матрицы  $\alpha$  для предоставления перестановок в рекурсивной функции  $R$ , служащей для получения стягивающего дерева глубины один (С.4), может быть заменено на, возможно, более компактное применение списка узлов, причем индексы для внутренних произведений представляются довольно очевидным, хотя и не совсем формальным способом. Кроме того,

такое рекурсивное описание может быть преобразовано в более эффективные нерекурсивные формы.

Наконец, любой алгоритм, ясно выраженный в терминах массивов, может быть преобразован простыми, хотя и утомительными модификациями в, по-видимому, более эффективные алгоритмы с применением итерации для скалярных элементов. Например, вычисление  $+/X$  зависит от каждого элемента из  $X$  и не допускает заметного улучшения, но вычисление  $\vee/B$  могло бы прекратиться на первом элементе, равном 1, и поэтому может быть улучшено итеративным алгоритмом, выраженным в терминах индексации.

Для математики весьма типично, когда сначала разрабатывается ясное и точное описание процесса безотносительно к эффективности, а затем оно используется в качестве руководства и теста для изучения эквивалентных процессов, обладающих другими характеристиками, например большей эффективностью. Это весьма плодотворная практика, которую не следует нарушать преждевременной заботой об эффективности компьютерного исполнения.

Оценки эффективности часто оказываются нереалистичными, потому что они принимают в расчет «основные» функции, например умножение и сложение, и пренебрегают вспомогательными (индексацией и другими процессами выбора), которые часто играют весьма важную роль в менее прямолинейных алгоритмах. Кроме того, реалистичные оценки в значительной мере зависят от современного проектирования компьютеров и от языковой реализации алгоритмов. Например, поскольку была обнаружена интенсивность использования в языке APL функций над булевыми значениями (таких, как  $\wedge/B$  и  $\vee/B$ ), разработчики обеспечили их эффективное выполнение. В конечном счете чрезмерное внимание к эффективности приводит к порочному кругу в проектировании: из соображений эффективности ранние языки программирования отражали характеристики ранних компьютеров и каждое новое поколение компьютеров отражало потребности языков программирования, разработанных для предыдущего поколения компьютеров.

#### БЛАГОДАРНОСТИ

Я благодарен моему коллеге А. Д. Фалкоффу за предложения, способствовавшие значительному улучшению организации этой статьи, а также профессору Д. Макинтайру за советы, возникшие у него при чтении чернового варианта рукописи.

## ПРИЛОЖЕНИЕ А

### СВОДКА НОТАЦИИ

$F\omega$	СКАЛЯРНЫЕ ФУНКЦИИ	$\alpha F\omega$
$\omega$	Сопряженная	+ Плюс
$0-\omega$	Отрицательная	- Минус
$(\omega > 0) - \omega < 0$	Знак	$\times$ Умножить
$1 \div \omega$	Обратная	$\div$ Разделить
$\lceil -\omega$	Модуль	Остаток
Целая часть	Низ	$P$ Минимум
$-\omega$	Вверх	$\lceil$ Максимум
$2.71828 \dots * \omega$	Экспоненциаль- ная	* Степень
Обратная для *	Натуральный логарифм	$\otimes$ Логарифм
$\times / 1 + i\omega$	Факториал	! Биноми- альная
$3.14159 \dots \times \omega$	Умножить на $Pi$	$\circ$

Булевы:  $\vee \wedge \sim$  (и, или, не-и, не-или, не)

Отношения:  $< \leq = \geq > \neq$  ( $\alpha R\omega$  равняется 1, если верно отношение  $R$ )

	Sec.	$V \leftrightarrow 2 \ 3 \ 5$	$M \leftrightarrow 1 \ 2 \ 3$
	Ref.		4 5 6
Integers	1	$15 \leftrightarrow 1 \ 2 \ 3 \ 4 \ 5$	
Shape	1	$\rho V \leftrightarrow 3$	$\rho M \leftrightarrow 2 \ 3$
Catenation	1	$V, V \leftrightarrow 2 \ 3 \ 5 \ 2 \ 3 \ 5$	$M, M \leftrightarrow 1 \ 2 \ 3 \ 1 \ 2 \ 3$
Ravel	1	$M \leftrightarrow 1 \ 2 \ 3 \ 4 \ 5 \ 6$	
Indexing	1	$V[3 \ 1] \leftrightarrow 5 \ 2$	$M[2; 2] \leftrightarrow 5$
Compress	3	$1 \ 0 \ 1 / V \leftrightarrow 2 \ 5$	$0 \ 1 / M \leftrightarrow 4 \ 5 \ 6$
Take, Drop	1	$2 + V \leftrightarrow 2 \ 3$	$-2 + V \leftrightarrow 1 + V \leftrightarrow 3 \ 5$
Reversal	1	$\phi V \leftrightarrow 5 \ 3 \ 2$	
Rotate	1	$2 \phi V \leftrightarrow 5 \ 2 \ 3$	$-2 \phi V \leftrightarrow 3 \ 5 \ 2$
Transpose	1, 4	$\mathbb{Q}\omega$ reverses axes	$\alpha \mathbb{Q}\omega$ permutes axes
Grade	3	$\Delta 3 \ 2 \ 6 \ 2 \leftrightarrow 2 \ 4 \ 1 \ 3$	$\nabla 3 \ 2 \ 6 \ 2 \leftrightarrow 3 \ 1 \ 2 \ 4$
Base value	1	$10 \vdash V \leftrightarrow 235$	$V \vdash V \leftrightarrow 50$
& inverse	1	$10 \ 10 \ 10 \vdash 235 \leftrightarrow 2 \ 3 \ 5$	$V \vdash 50 \leftrightarrow 2 \ 3 \ 5$
Membership	3	$V \in 3 \leftrightarrow 0 \ 1 \ 0$	$V \in 5 \ 2 \leftrightarrow 1 \ 0 \ 1$
Inverse	2, 5	$\mathbb{Q}\omega$ is matrix inverse	$\alpha \mathbb{Q}\omega \leftrightarrow (\mathbb{Q}\omega) + . \times \alpha$
Reduction	1	$+ / V \leftrightarrow 10$	$+ / M \leftrightarrow 6 \ 15$
Scan	1	$+ \backslash V \leftrightarrow 2 \ 5 \ 10$	$+ \backslash M \leftrightarrow 2 \ 3 \rho 1 \ 3 \ 6 \ 4 \ 9 \ 15$
Inner prod	1	$+ . \times$ is matrix product	
Outer prod	1	$0 \ 3 \circ + 1 \ 2 \ 3 \leftrightarrow M$	
Axis	1	$F[I]$ applies $F$ along axis $I$	

## ПРИЛОЖЕНИЕ Б

### КОМПИЛЯТОР ИЗ НЕПОСРЕДСТВЕННОЙ В КАНОНИЧЕСКУЮ ФОРМУ

Этот компилятор был заимствован из [22, с. 222]. Он не будет обрабатывать описания, включающие  $\alpha$ , или  $\omega$  в кавычках. Он состоит из функций *FIX* и *F9* и матриц символов *C9* и *A9*:

*FIX*

$0\rho\Box FX\ F9\ \Box$

$D\leftarrow F9\ E;F;I;K$

$F\leftarrow(,(E=' \omega ') \circ . \neq 5+1) / , E, (\phi 4, \rho E) \rho ' \ Y9\ '$

$F\leftarrow(,(F=' \alpha ') \circ . \neq 5+1) / , F, (\phi 4, \rho F) \rho ' \ X9\ '$

$F\leftarrow 1+\rho D\leftarrow(0,+/^{-}6,I)+(- (3\times I))+\backslash I\leftarrow ': '=F)\phi F, (\phi 6, \rho F) \rho ' \cdot$

$D\leftarrow 3\phi C9[1+(1+' \alpha '\in E), I, 0;], \phi D[;1, (I+2+1F), 2]$

$K\leftarrow K+2\times K<1\phi K\leftarrow I\wedge K\in(> / 1\ 0\phi '\leftarrow \Box ' \circ . =E) / K\leftarrow +\backslash \sim I\leftarrow E\in A9$

$F\leftarrow(0,1+\rho E)\lceil \rho D\leftarrow D, (F, \rho E)\rceil \phi 0\ ^{-}2+K\phi ' \ , E, [1.5]';'$

$D\leftarrow (F\leftarrow D), [1]F[2]\ ' \ A', E$

*C9*

$Z9\leftarrow$

$Y9Z9\leftarrow$

$Y9Z9\leftarrow X9$

$) / 3\leftarrow (0=1\leftarrow ,$

$\rightarrow 0, 0\rho Z9\leftarrow$

*A9*

012345678

9ABCDEFGHI

IJKLMNOPQ

RSTUVWXYZ

ABCDEFGHI

JKLMNOPQR

STUVWXYZ $\Box$

Пример:

*FIX*

$FIB:Z, + / ^{-}2+Z\leftarrow FIB\omega-1: \omega=1:1$

*FIB 15*

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610

$\Box CR' FIB'$

$Z9\leftarrow FIB\ Y9; Z$

$\rightarrow (0=1\leftarrow , Y9=1) / 3$

$\rightarrow 0, 0\rho Z9\leftarrow 1$

$Z9\leftarrow Z, + / ^{-}2+Z\leftarrow FIB\cdot Y9-1$

$\mathbf{A}FIB:Z, + / ^{-}2+Z\leftarrow FIB\omega-1: \omega=1:1$

### ЛИТЕРАТУРА

1. Boole G. An Investigation of the Laws of Thought, Dover Publication, N. Y., 1951. Первая публикация в 1954 г. Walton and Maberly, London and by MacMillan and Co. Cambridge. См. также: Collected Logical Works of

- George Boole, Volume 2, Open Court Publishing Co., La Salle, Illinois, 1916.
2. Cajori F. A. A History of Mathematical Notations, Volume 2, Open Court Publishing Co., La Salle, Illinois, 1929.
3. Falkoff A. D. and Iverson K. E. The Evolution of APL, Porceedings of a Conference on the History of Programming Languages, ACM SIGPLAN, 1978.
4. APL Language, Form No. GC26—3847—4, IBM Corporation.
5. Iverson K. E. Elementary Analysis, APL Press, Pleasantville, N. Y., 1976.
6. Iverson K. E. Algebra: An Algorithmic Treatment, APL Press, Pleasantville, N. Y., 1972.
7. Kerner I. O. Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen, Numerische Mathematik, Vol. 8, 1966, pp. 290—294.
8. Beckenbach E. F., ed. Applied Combinatorial Mathematics., John Wiley and Sons, New York, N. Y., 1964.
9. Tarian R. E. Testing Flow Graph Reducibility, Journal of Computer and Systems Sciences, Vol. 9, No. 3, Dec. 1974.
10. Spence R. Resistive Circuit Theory, APL Press, Pleasantville, N. Y., 1972.
11. Iverson K. E. A Programming Language, John Wiley and Sons, New York, N. Y., 1962.
12. Iverson K. E. An Introduction to APL for Scientists and Engineers, APL Press, Pleasantville, N. Y.
13. Orth D. L. Calculus in a New Key, APL Press, Pleasantville, N. Y., 1976.
14. Apoctol T. M. Mathematical Analysis, Addison Wesley Publishing Co., Reading, Mass., 1957.
15. Jolley L. B. Summation of Series, Dover Publications, N. Y.
16. Oldham K. B. and Spanier J. The Fractional Calculus, Academic Press, N. Y., 1974.
17. APL Quote Quad, Vol. 9, No. 4, June 1979, ACM STAPL.
18. Iverson K. E. Operators and Functions, IBM Research Report RC 7091, 1978.
19. McDonnel E. E. A Notation for the GCD and LCM Functions, APL 75, Proceeding of an APL Conference, ACM, 1975.
20. Iverson K. E. Operators, ACM Transactions on Programming Languages and Systems, October 1979.
21. Blaauw G. A., Digital System Implementation, Prentice-Hall, Englewood Cliffs, N. J., 1976.
22. McIntyre D. B. The Architectural Elegance of Crystals Made Clear by APL, An APL Users Meeting, I. P. Sharp Associates, Toronto, Canada, 1978.

## ПОСТСКРИПТУМ

### НОТАЦИЯ КАК СРЕДСТВО МЫШЛЕНИЯ: 1986

К. Е. Айверсон

Тезис данной работы состоит в том, что обнаруживаемые в языках программирования исполнимость и универсальность могут быть эффективно объединены в одном согласованном языке с преимуществами, предоставляемыми математической нотацией.

В качестве исполнимого языка мы будем применять APL, язык общего назначения, который был создан с целью обеспечения ясного и точного выражения мыслей при написании и преподавании, который был реализован в виде языка программирования только после нескольких лет его разработки и развития.

Выше приведены две цитаты из моей статьи 1980 г. В первой из них указывается, в каком случае следует использовать исполнимую аналитическую нотацию, а во второй указано конкретное средство для ее разработки.

Наиболее очевидным и важным является применение исполнимой аналитической нотации в преподавании. В последующих комментариях подводятся итоги недавнего прогресса в этой области.

## ПРЕДМЕТЫ И КУРСЫ

Общей темой упомянутых здесь предметов является неформальное введение необходимой нотации в контекст способом, знакомым по обучению математике. Хорошим примером на уровне высшей школы может служить подход к теории вероятности, использованный в [1]. В [2] язык APL широко используется как средство реализации системы для анализа схем, а в [3] графический ввод и выражения языка APL совместно применяются для описания проектов в экспертной системе.

Направление моей недавней работы описано в письме [4], и теперь доступны наброски двух текстов [5], используемых в курсах. Всем, кто интересуется этими вопросами, следует прочесть статью [6].

## РАЗВИТИЕ НОТАЦИИ

Недавно была разработана версия языка APL [7], которая, оставаясь в рамках стандарта ISO для этого языка, упростила его структуру и увеличила его выразительную силу. Она обеспечивает значительно лучшую основу для преподавания, чем нотация, использованная в моей статье 1980 г.

## ДОСТУПНОСТЬ РЕАЛИЗАЦИЙ

Хотя APL в течение долгого времени предлагался централизованными вычислительными службами университетов, его применение в преподавании оказывалось непрактичным из-за высоких затрат и отсутствия подходящих терминалов. В наше время доступность систем APL на микрокомпьютерах резко изменила эту ситуацию.

Я считаю наиболее удовлетворительной систему, предоставляемую нашим студентам [8]; в ней еще не воплощены такие новые функции, как **сущность**, **стереть** и **все** (обобщение декартова произведения), но предусмотрены фундаментальные понятия **ранга функции**, **блочной функции** (для общей обработки представлений или «структур») и операция **под** для важного математического понятия двойственности.

Кроме того, система оперирует комплексными числами (со всеми математическими функциями, подходящим образом обобщенными), обеспечивает детерминант ( $-X$ ), перманент ( $+X$ ), проверку для латинского квадрата ( $V.A$ ) и связанные с ними функции, порождаемые операцией «точка», обобщения функций и **или** для получения наибольшего общего делителя и наименьшего общего кратного; используются характеристики микрокомпьютера и экрана его монитора для обеспечения «единой» клавиатуры, в которой большинство символов (в частности, скобки и прописные и строчные буквы, употребляемые в именах) находятся в своих обычных позициях, принятых на пишущих машинках.

## ЛИТЕРАТУРА

1. Alvord L. Probability in APL. APL press, STSC Corp., Bethesda Md.
2. Spence R. and Burgess J. Circuit Analysis. Prentice-Hall, Englewood Cliffs, N. J., 1986.
3. Hasony Y. Краткий отчет о его работе появился в обзоре конференции Minnowbrook, представленной в APL Quote-Quad 16, 3 (1986).

4. Blaauw G. A. et al. A curriculum proposal for computer science. Commun. ACM, Forum (Sept. 1985).
5. Iverson K. E. Mathematics and Programming; Applied Mathematics for Programmers. Обе работы можно получить из I. P. Sharp Associates, Toronto, Ont., Canada.
6. Pesch R. and Berry M. J. A Style and Literacy in APL. In Proceedings of APL86, ACM, New York, 1986.
7. Iverson K. E. A Dictionary of the APL Language. Можно получить из I. P. Sharp Associates, Toronto, Ont., Canada.
8. Sharp APL/PCX. Computer system for use on IBM AT/370 and XI/370 computers. Можно получить из I. S. Sharp Associates, Toronto, Ont., Canada. Система работает также на обычных IBM PC или AT (значительно более медленно, но приемлемо для учебных целей).

# Реляционная база данных: практическая основа эффективности

Э. Д. Кодд

Исследовательская лаборатория фирмы IBM в Сан-Хосе  
(IBM San Jose Research Laboratory)

Премия Тьюринга 1981 г. Ассоциации вычислительных машин (АСМ) была вручена президентом Питером Деннингом 9 ноября 1981 г. Эдгару Ф. Кодду, сотруднику Исследовательской лаборатории фирмы IBM в Сан-Хосе, на ежегодной конференции АСМ. Конференция состоялась в Лос-Анджелесе, Калифорния. Это главная премия Ассоциации, присуждаемая за выдающийся технический вклад в информатику.

Генеральный комитет АСМ по премиям за технические достижения выбрал Кодда за его «продолжительный фундаментальный вклад в теорию и практику развития СУБД». Создатель реляционной модели баз данных, Кодд много сделал для развития реляционной алгебры (алгебры отношений), реляционного исчисления и нормализации отношений (normalization of relations).

Эдгар Кодд поступил на фирму IBM в 1949 г. Его задачей было писать программы для системы, называемой Selective Sequence Electronic Calculator. С тех пор его работа в области информатики была связана с логическим проектированием компьютеров (IBM701 и Stretch). Он руководил компьютерным центром в Канаде, возглавлял создание одной из первых операционных систем с возможностью мультипрограммного режима работы, занимался логикой самовоспроизводящихся автоматов, развивал методы высокого уровня для технических условий на средства программирования (software specifications), создавал и совершенствовал реляционный подход к управлению базами данных и работал над подсистемой анализа и синтеза английской речи для случайных пользователей баз данных. Он также является автором «Клеточных автоматов», ранней книги из серии монографий, издаваемых Ассоциацией вычислительных машин.

Кодд получил степени бакалавра и магистра искусств по математике в Оксфордском университете в Англии и степени магистра естественных наук и доктора философии в области вычислительной техники и информатики в Мичиганском университете. Он является членом Национальной академии инженерных наук США (National Academy of Engineering (USA))



и Британского компьютерного общества (British Computer Society).

Премия Тьюринга АСМ присуждается ежегодно в честь А. М. Тьюринга, английского математика, чьи исследования явились большим вкладом в информатику.

Хорошо известно, что увеличивающийся спрос конечных пользователей на новые приложения превосходит возможности отраслей, занимающихся обработкой информации, по созданию прикладных программ. Существует два взаимно дополняющих подхода к решению этой проблемы (и оба они являются необходимыми): один заключается в том, чтобы дать возможность конечным пользователям напрямую взаимодействовать с информацией, хранящейся в памяти компьютера; второй состоит в увеличении производительности труда профессионалов в области обработки данных при создании прикладных программ. Менее известно, что один-единственный метод — управление реляционными базами данных — дает практическую основу для обоих подходов. В данной статье объяснено, почему это так.

При обсуждении вопросов производительности отмечено, что пора провести четкую границу между реляционными и нереляционными базами данных, чтобы правильное истолкование термина «реляционная» не вводило в заблуждение. Ключом к тому, где проводить эту границу, является нечто, называемое «возможностью работы с реляционной моделью данных».

## 1. ВВЕДЕНИЕ

Общеизвестно, что сейчас происходит «кризис производительности» в области создания пользовательских программ («running code») для коммерческих и промышленных применений. Отрасли, занимающиеся обработкой информации, уже неспособны обеспечить прикладными программами все увеличивающиеся потребности конечных пользователей в новых приложениях. В конце семидесятых — начале восьмидесятых годов многие люди, работающие в области информатики, надеялись, что введение систем управления базами данных (для их обозначения принято сокращение СУБД) значительно повысит эффективность работы прикладных программистов, устранив многочисленные проблемы, возникающие при работе с входными и выходными файлами. Оказалось, что СУБД (вместе со словарями данных) являются очень эффективным средством управления данными, и они действительно избавили программистов от множества проблем, возникающих при обработке файлов. Почему все же не удалось увеличить производительность?

Существуют три основные причины этого:

(1) Эти системы обременили прикладных программистов многочисленными понятиями, не имеющими прямого отношения к задачам поиска и обработки информации, вынуждая их думать и писать на неоправданно низком уровне структурных подробностей (наглядным примером этого являются «владель-

цы и члены множества»<sup>1)</sup>, которых придумали члены группы по созданию базы данных CODASYL);

(2) Не было предусмотрено средств для одновременной обработки нескольких записей — другими словами, СУБД не поддерживали работу с множествами, и в результате программисты были вынуждены думать и писать на языке циклов итераций, что часто было вовсе не обязательно (здесь мы используем слово «множество» в его традиционном математическом смысле, а не в смысле связной структуры, как подразумевает рабочая группа о базе данных CODASYL);

(3) Тот факт, что конечному пользователю потребуется непосредственное взаимодействие с базами данных, особенно непредсказуемое по своему характеру взаимодействие, был неправильно истолкован: считалось, что возможность работы с запросами — это то, что можно добавить к СУБД когда-нибудь потом.

Оглядываясь на системы управления базами данных, созданные в конце шестидесятых годов, можно сразу заметить, что в них не было существенного различия между представлением об информации программиста (логическим) и представлением информации в памяти машины (физическим). Если даже при так называемом логическом представлении защита от размещения осуществлялась на языке адресов памяти и побайтовых сдвигов, то и многие понятия, ориентированные на работу с памятью, были неотъемлемой частью того же уровня представлений.

Требование к программистам ориентироваться в путях доступа, чтобы достигнуть целевой информации (в некоторых случаях им приходилось иметь дело непосредственно с представлением данных в памяти, а в других — следовать по цепочке указателей) оказало очень неблагоприятное влияние. Ко всему прочему любое небольшое изменение размещения в памяти приводило к немедленной переделке всех программ, использующих предыдущую структуру. Введение индекса (предметного указателя) могло привести к тому же эффекту. В результате слишком много человеческих сил было истрачено на постоянную поддержку прикладных программ, которой можно было бы избежать.

---

<sup>1)</sup> Причина трудностей, связанных с этой концепцией, состоит в том, что она объединяет в одной конструкции три независимых понятия: отношение «один — несколько», зависимость от существования объекта и видимая пользователю связная структура, вдоль которой должны пролагать свои маршруты прикладные программы. Именно последнее из этих понятий налагает на прикладных программистов тяжелое и ненужное бремя ориентирования в путях доступа, а также создает непреодолимые препятствия для конечных пользователей.

Другое следствие состояло в том, что ввод в эксплуатацию таких систем в большинстве случаев происходил очень медленно, потому что прежде чем активировать базу данных, много времени уходило на обучение работе в системе и на планирование организации данных как на логическом, так и на физическом уровне. Целью этого предварительного планирования было «сделать правильно сразу и навсегда», чтобы избежать необходимости впоследствии изменять описания данных, что в свою очередь повлечет изменения в прикладных программах. Такая цель, конечно, была недостижима, даже если бы основные принципы построения баз данных были известны в то время (а они, конечно, не были известны).

Чтобы показать, как системы управления реляционными базами данных избегают трех перечисленных выше ловушек, мы вначале рассмотрим причины, побудившие создавать реляционную модель данных, и обсудим некоторые ее черты. Потом мы займемся классификацией систем, основанных на этой модели. По мере продвижения мы будем обращать внимание на работу прикладного программиста, хотя выгоды для конечного пользователя также очень велики. О значении реляционных баз данных уже много говорилось и было приведено много примеров (см. [23] и ссылки в этой работе).

## 2. МОТИВИРОВКА

Основной побудительной причиной исследований, результатом которых стало создание реляционной модели данных, было желание четко разграничить логический и физический аспекты управления базами данных (принимая во внимание проблемы создания баз данных, поиска и обработки информации). Мы назвали это *стремлением к независимости данных*.

Вторым нашим желанием было создать структурно простую модель, так чтобы пользователи и программисты любой квалификации одинаково могли бы понимать содержащуюся в ней информацию и могли бы поэтому общаться друг с другом при помощи базы данных. Мы назвали это *стремлением к коммуни-кабельности*.

В-третьих, мы хотели использовать концепции языка высокого уровня (но не специфический синтаксис), чтобы пользователи имели возможность описывать операции сразу над большими порциями информации. Это является основой для способов обработки информации, ориентированных на множества (т.е. возможности при помощи одного оператора задать операцию над несколькими множествами записей одновременно). Мы назвали это *стремлением к обработке множеств*.

У нас также были и другие цели, такие как разработка тео-

ретических основ организации и управления базами данных, однако они не имеют непосредственного отношения к нашей теме эффективности.

### 3. РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ

Чтобы реализовать все три стремления, было необходимо отказаться от всех тех понятий структуризации данных (например, повторяющихся групп, связанных структур), которые не известны конечным пользователям, и выработать новый взгляд на адресацию информации.

Понятие позиционирования всегда играло важную роль при работе с адресами в памяти компьютера, начиная со способа адресации с помощью наборного поля (коммутационной панели), потом абсолютной численной адресации, относительной численной адресации и символической адресации с арифметическими свойствами (как, например, символический адрес вида  $A+3$  в ассемблере; адрес  $X(I+1, J-2)$  элемента массива  $X$  в Алголе, Фортране, PL/I). В реляционной модели мы заменяем позиционный способ адресации общим ассоциативным способом.

Каждая единица информации в реляционной базе данных имеет уникальный адрес в виде имени отношения (relation name), значения первичного ключа данных и имени атрибута. Ассоциативный способ адресации такого вида позволяет пользователям (да-да, и даже программистам) предоставить системе самой (1) детально определять, где разместить новый фрагмент информации, вводимый в базу данных, и (2) выбирать соответствующие пути доступа при поиске данных.

Вся информация в реляционной базе данных представляется в виде значений в таблицах (даже имена таблиц являются символьными строками по крайней мере в одной таблице). Адресация информации по значению, а не по позиции, увеличивает эффективность работы как программистов, так и конечных пользователей (позиция элемента в последовательности обычно подвергается изменениям, и за ней довольно трудно следить, особенно если последовательность содержит много элементов). Более того, тот факт, что как программисты, так и конечные пользователи используют одинаковый способ адресации, дает возможность реализовать свойство коммуникативности.

В качестве структурной единицы для реляционной модели данных было выбрано отношение  $n$ -го порядка, потому что при помощи соответствующих операторов и соответствующего концептуального представления (таблицы) оно ведет к реализации всех трех свойств, перечисленных выше. Заметим, что отноше-

ние  $n$ -го порядка — это математическое множество, в котором порядок строк не имеет значения.

Иногда возникают следующие вопросы: Почему такая модель называется реляционной моделью данных? Почему бы не назвать ее табличной? Для этого есть две причины: (1) Когда реляционная модель данных была предложена впервые, многие люди, занимающиеся обработкой данных, считали, что отношение между двумя или более объектами нужно представлять с помощью связанных структур данных (и название было дано так, чтобы учесть это неправильное представление); (2) понятие таблицы находится на более низком уровне абстракции, чем понятие отношения, так как при его использовании создается впечатление, что применим способ адресации по позиции (в то время как это неверно для отношений  $n$ -го порядка), и тот факт, что содержание информации в таблице не зависит от порядка строк, не является очевидным. Тем не менее, даже обладая этими несущественными недостатками, таблицы являются основным средством концептуального представления отношений, потому что слово «таблица» понятно всем.

В тех случаях, когда какая-либо модель данных рассматривается как серьезная альтернатива реляционной модели, она также должна иметь четко определенное концептуальное представление для того, чтобы на ее основе создать базу данных. Наличие такого представления упрощает размышления о предполагаемых операциях. Это необходимо для эффективной работы программистов и конечных пользователей. В моделях данных, в которых используются такие понятия, как объекты (entities) и отношения, или в функциональных моделях данных такое представление если и обсуждается, то очень редко. В таких моделях часто вообще нет операторов! Тем не менее такие модели могут быть полезны при некоторых способах анализа типов данных, который приходится делать при создании новых баз данных, и на совсем ранних этапах работы, когда предварительно на неформальном уровне определяются способы ее организации. Все это приводит к вопросу: а что же это такое — модель данных?

Модель данных — это, конечно, не структура данных, как, возможно, думают многие. Естественно, что основные модели данных называются в соответствии с их основными структурами данных, но это еще не все.

Модель данных [9] — это комбинация по крайней мере трех составляющих:

(1) Набора типов структур данных (являющихся блоками при построении базы данных);

(2) Набора операторов или правил вывода, которые могут быть применены к любым правильным примерам типов дан-

ных, перечисленных в (1), чтобы находить, выводить или преобразовывать информацию, содержащуюся в любых частях этих структур в любых комбинациях.

(3) Набора общих правил целостности, которые прямо или косвенно определяют множество непротиворечивых состояний базы данных, или множество изменений ее состояния, или и то и другое. Эти правила являются общими в том смысле, что они применяются к любой базе данных, использующей данную модель. (Кстати, иногда они могут выражаться в виде правил ввода-обновления-вывода.)

Реляционная модель является моделью данных в этом смысле, и она является первой, удовлетворяющей этому определению. Мы не считаем нужным приводить в этой статье подробное определение реляционной модели — первое определение появилось в работе [7], затем оно было уточнено во второй и третьей частях работы [8]. Структурная часть реляционной модели данных состоит из доменов, отношений неопределенного порядка (relations of assorted degrees) (основным средством концептуального представления которых являются таблицы), атрибутов, кортежей (tuples), потенциальных ключей и первичных ключей. В соответствии с выбранным представлением атрибуты становятся столбцами таблиц, а кортежи — строками, но здесь, когда это касается таблиц нашей базы данных, не существует понятия того, что один столбец таблицы следует за другим или одна строка следует за другой. Другими словами, в этих таблицах порядок столбцов слева направо и порядок строк сверху вниз является произвольным и несущественным.

Обрабатывающая часть реляционной модели данных состоит из алгебраических операторов (выбора <select>, проекции <project>, соединения <join> и т. д.), которые преобразуют отношения в отношения (и, следовательно, таблицы в таблицы).

Часть, касающаяся целостности, состоит из двух правил — целостности объекта (entity integrity) и целостности на уровне ссылок (referential integrity) (последние достижения в этой области описаны в [8, 11]). В любой конкретной реализации модели данных может оказаться необходимым наложить дополнительные (зависящие от базы данных) ограничения, обеспечивающие целостность, и таким образом определить меньшее множество непротиворечивых (совместимых, значимых) состояний базы данных или их изменений.

В процессе развития реляционной модели данных всегда соблюдалось строгое соответствие между структурными аспектами и аспектами обработки и целостности. Если бы структуры создавались сами по себе, независимо от всего остального, свойства их поведения ничем не были бы связаны, и возникали бы бесконечные возможности и бесконечно продолжающиеся рас-

суждения. Поэтому неудивительно, что попытки, которые принимаются группами CODASYL и ANSI, развивать язык определения данных (DDL) и язык обработки данных (DML) в различных комитетах породили множество недоразумений и несовместимых результатов.

#### 4. ВОЗМОЖНОСТЬ РЕЛЯЦИОННОЙ ОБРАБОТКИ ИНФОРМАЦИИ

Для реляционной модели данных необходимы не только реляционные структуры данных (которые можно считать таблицами), но также и определенный способ работы с множествами, который называется *реляционной обработкой данных*. При реляционной обработке данных приходится иметь дело с отношениями как с операндами. Ее основными задачами является предотвращение заикливания (loop-avoidance), абсолютное требование эффективности для конечного пользователя и значительное увеличение производительности работы прикладных программистов.

Оператор реляционной алгебры SELECT (также иногда называемый RESTRICT) берет в качестве операнда *одно* отношение (таблицу) и преобразует его в новое отношение (таблицу), состоящее из выбранных кортежей (строк) первого отношения. Оператор PROJECT также преобразовывает *одно* отношение (таблицу) в новое, состоящее на этот раз из выбранных атрибутов (столбцов) первого. Оператор EQUI—JOIN берет в качестве операндов *два* отношения (две таблицы) и преобразует их в третью таблицу, строки которой состоят из строк первой таблицы, сцепленных со строками второй, но только в тех местах, где определенные столбцы первой и второй таблиц имеют совпадающие значения. Если исключаются избыточные столбцы, соответствующий оператор называется NATURAL JOIN. Впоследствии мы будем использовать термин «соединение» (join) для обозначения операторов соединения (equi-join) или естественного соединения (natural join).

Реляционная алгебра, включающая в себя эти и другие операторы, должна быть критерием производительности системы. Она не предназначена для того, чтобы быть стандартным языком, которого обязаны придерживаться все реляционные системы. Способность реляционной системы к обработке множеств необходимо поддерживать средствами подязыка манипулирования данными<sup>1)</sup>, который по меньшей мере должен обладать

---

<sup>1)</sup> Подязык манипулирования данными — это специализированный язык для управления базой данных, поддерживающий по крайней мере операции определения (описания), поиска, ввода, обновления и удаления информации. Он не должен удовлетворять требованию вычислительной полноты и обычно

эффективностью реляционной алгебры, *не используя при этом итерационных и рекурсивных выражений*.

Мощность системы вывода, созданной на базе реляционной алгебры, определяется в основном только операторами SELECT, PROJECT и JOIN, при условии что оператор JOIN не подвергается таким ограничениям при реализации, при которых он должен иметь дело с предварительным определением соответствующих физических путей доступа. Система обладает *неограниченной способностью к соединению*, если она допускает соединения в любой паре сопоставимых атрибутов, при одном условии, что они определены на одном домене или типе данных (для наших целей неважно, является ли домен синтаксическим или семантическим и является ли тип данных слабым или сильным, однако в работе [10] описаны обстоятельства, при которых это существенно).

Иногда встречаются системы, в которых операция соединения выполняется только при условии, что сравниваемые атрибуты имеют одинаковое имя или поддерживаются заранее предопределенным путем доступа определенного типа. Такие ограничения значительно уменьшают способность системы к выводу новых отношений из основных. Следовательно, такие ограничения уменьшают возможности системы обрабатывать непредусмотренные запросы конечных пользователей и уменьшают шансы прикладного программиста избежать заикливания.

Таким образом, мы говорим, что субязык манипулирования данными  $L$  обладает *возможностью реляционной обработки*, если преобразования, определенные операторами реляционной алгебры SELECT, PROJECT и неограниченным оператором JOIN могут быть определены в  $L$  без использования итераций или рекурсии. Для того, чтобы система управления базой данных называлась *реляционной*, она должна поддерживать:

- (1) Таблицы без видимых пользователю навигационных связей между ними;
- (2) Подязык манипулирования данными, обеспечивающий по крайней мере эту (минимальную) возможность реляционной обработки.

Из этого в первую очередь следует, что СУБД, которая *не* поддерживает реляционную обработку, должна рассматриваться как *нереляционная*. Более подходящим названием для такой системы является «табличная» при условии, что она поддерживает работу с таблицами без видимых пользователю навигационных путей между таблицами. Следовало бы заменить этим термином термин «полуреляционная», использованный

не удовлетворяет ему. С точки зрения программирования прикладных систем он предназначен для использования вместе с одним или несколькими языками программирования.



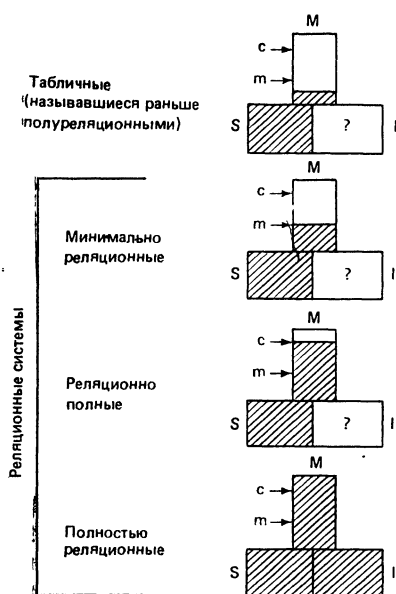


Рис. 1. Классификация СУБД:  $S$  — структурная часть,  $M$  — обрабатывающая часть;  $I$  — целостность,  $c$  — реляционная полнота,  $t$  — минимальная возможность реляционной обработки.

тов и на уровне ссылок). Реляционная система, в которой в полной мере реализованы требования к двум последним частям реляционной модели данных, называется *полностью реляционной* [8]. Хотя в настоящее время не существует баз данных, которые можно было бы назвать полностью реляционными, многие системы уже очень близки к этому и нет сомнений, что очень скоро такие базы данных появятся.

На рис. 1 продемонстрированы различия между разными классами реляционных и табличных систем. Для каждого класса величина заштрихованной части прямоугольника  $S$  показывает, насколько в базах данных из этого класса выполняются структурные требования, предъявляемые к реляционной модели данных. То же самое относится и к прямоугольнику  $M$  и требованиям к обрабатывающей части базы данных, и к прямоугольнику  $I$  и требованиям к части базы данных, касающейся правил целостности.

Буква  $t$  соответствует минимальной возможности реляционной обработки, буква  $c$  означает реляционную полноту (т. е. таким возможностям системы, которые соответствуют логике

в работе [8], потому что существует большая разница в сложности реализации между табличными системами, в которых программист строит свои навигационные пути, и реляционными системами, которые самостоятельно строят пути доступа для программиста, т. е. системами, обеспечивающими *автоматическую навигацию*.

Данное выше определение реляционной СУБД допускает большую свободу при определении предоставляемых услуг. Например, вовсе не требуется, чтобы реляционная алгебра была реализована в полном объеме, и не предъявляется никаких требований, относящихся к поддержке двух правил целостности реляционной модели данных (целостности на уровне объектов

двуместных предикатов первого порядка без нулей. Если прямоугольник **М**, обозначающий обрабатываемую часть базы данных, заштрихован полностью, это значит, что в базе данных в полной мере реализованы возможности, соответствующие реляционной алгебре, определенной в [8] (возможности системы соответствуют логике трехместных предикатов с нулем одного типа). Для всех классов систем на рис. 1, кроме класса полностью реляционных баз данных, в прямоугольниках, обозначающих часть, реализующую правила целостности, стоит вопросительный знак. Это показывает, что в настоящее время в таких системах отсутствует адекватная реализация свойств целостности. Необходимы также более эффективная поддержка доменов и первичных ключей [10] и специальные средства реализации, описанные в работе [14].

Заметим, что реляционные СУБД могут сочетать возможности реляционной обработки любым удобным способом. Например, в системе INGRES фирмы Relational Technology, Inc. операция RETREIVE языка запросов QUEL [29] содержит все три оператора (выбора  $\langle \text{select} \rangle$ , проекции  $\langle \text{project} \rangle$  и соединения  $\langle \text{join} \rangle$ ) в одном запросе, так что можно получить тот же самый результат, что и при применении каждого из этих операторов по отдельности или любого их сочетания.

Определение реляционной модели данных содержит несколько запрещений. Приведем два примера: исключены видимые для пользователя навигационные связи между таблицами, и информация, содержащаяся в базе данных, не должна быть представлена (или скрыта) порядком corteжей в отношениях. Наш опыт говорит о том, что разработчики СУБД, которые имели дело с нереляционными системами, с трудом понимают и принимают эти ограничения. Пользователи же, наоборот, с энтузиазмом воспринимают значительное упрощение обучения и использования, которое является следствием этих запрещений.

Кстати, группа по реляционным системам Американского института стандартов (Relational Task Group of the American National Standards Institute) недавно опубликовала доклад о возможности создания стандарта для реляционных баз данных [4]. В этой работе содержится поучительный анализ доброй дюжины реляционных систем, и ее авторы хорошо понимают, что такое реляционная модель данных.

## 5. СВОЙСТВО УНИВЕРСАЛЬНОЙ РЕЛЯЦИОННОСТИ

Для расширения области применения в большинстве реляционных баз данных существуют подязыки описания данных (манипулирования данными), которые могут иметь интерфейс

с одним или несколькими широко распространенными языками программирования (например, Коболом, Фортраном, PL/I, APL). Мы будем в дальнейшем называть последние *базовыми* языками. Реляционная СУБД обычно поддерживает по меньшей мере один подязык описания данных, ориентированный на конечного пользователя, а иногда и несколько таких подязыков, потому что потребности пользователей могут различаться. Одни предпочитают описательные языки (string languages), такие как QUEL или SQL [5], другим больше нравится экранный двумерный подязык управления данными Query-by-Example (язык запросов на примерах) [33].

В настоящее время некоторые реляционные системы (например, System R [6], INGRES [29]) поддерживают языки описания данных, которые могут использоваться в двух режимах: (1) интерактивном, который необходим при работе с терминалом, и (2) в режиме «встроенности» в прикладную программу, написанную на базовом языке. Существуют серьезные причины возникновения подязыков описания данных с *двумя режимами*:

(1) С помощью такого языка прикладной программист может независимо отлаживать с терминала только обращения к базе данных, которые он предполагает включить в свои программы. Программисты, использовавшие при работе SQL, считают, что наличие двух режимов значительно повышает эффективность их работы (скорость отладки программ);

(2) Язык такого типа значительно увеличивает коммуникационные возможности СУБД для программистов, аналитиков, конечных пользователей, персонала администрации базы данных и т. д.;

(3) Произвольные различия между разными языками, используемыми в двух режимах, излишне обременяют память тех пользователей, которым приходится работать в обоих режимах, и требуют дополнительных усилий при обучении.

Наличие двух режимов в языке описания данных является настолько важным для эффективности системы, что существует классификация реляционных СУБД в зависимости от того, обладают они этим свойством или нет. Мы называем реляционные базы данных, в которых поддерживается подязык описания данных с двумя режимами, *универсально реляционными*. Таким образом, универсально реляционные СУБД поддерживают реляционную обработку как на уровне интерфейса с конечным пользователем, так и на уровне интерфейса с прикладным программистом *с помощью подязыка описания данных, общего для обоих интерфейсов*.

Естественным названием для всех остальных СУБД является «*неуниверсально реляционные*». Примером неуниверсально

реляционной базы данных является TANDEM ENCOMPASS [19]. В этой системе при интерактивном режиме поиска информации с терминала используется реляционный подязык описания данных ENFORM (это язык, обладающий возможностями реляционной обработки). При написании программы, в которой требуется найти или обработать информацию, используется расширенная версия языка Кобол (языка, который не обладает возможностями для реляционной обработки). На обоих уровнях используются общие структуры: таблицы без видимых пользователю навигационных путей между ними.

Немедленно возникает следующий вопрос: каким образом подязык, обладающий возможностями реляционной обработки, сопрягается с такими языками, как, например, Кобол или PL/I, которые могут обрабатывать записи только по одной (т. е. не обладают возможностью воспринимать множество записей как один операнд). Чтобы решить эту проблему, необходимо различать два действия: (1) определение отношения, которое выводится; (2) представление выведенного отношения в программе, написанной на базовом языке.

Одно решение (принятое в системе Peterlee Relational Test Vehicle [31]) состоит в том, чтобы представлять выведенное отношение в виде файла, который может быть прочитан запись за записью с помощью операторов базового языка. В этом случае записи выдаются в файл той файловой системы, которую используют конкретный базовый язык.

Другое решение (принятое в системе System R) состоит в том, чтобы управлять выдачей записей с помощью предложений подязыка описания данных и, следовательно, оптимизатора реляционной СУБД. Оператор запроса Q языка SQL (подязыка описания данных системы System R) может быть помещен в программу, написанную на базовом языке, в виде следующего предложения (для наглядности его синтаксис не совсем точно совпадает с синтаксисом SQL):

DECLARE C CURSOR FOR Q

где C — любое имя, выбранное программистом. Такое предложение ставит в соответствие (ассоциирует) курсор с именем C и определение выражения Q. Кортжи из выведенного отношения, определенного запросом Q, предъявляются программе по одному с помощью курсора с именем. Каждый раз, когда посредством этого курсора выполняется операция выборки FETCH, система выдает следующий кортеж выведенного отношения. Порядок выдачи определяется системой, за исключением тех случаев, когда запрос Q, определяющий выведенное отношение, содержит условие ORDER BY.

Важно отметить, что продвигая курсор по выведенному от-

ношению, программист не участвует в процессе навигации по направлению к некоторой целевой информации. Полученное отношение само и является целевой информацией! Именно СУБД сама определяет, должно ли выведенное отношение быть материализовано целиком перед тем, как просматривать его с помощью курсора, или по частям в процессе просмотра. В любом случае именно система (а не программист) выбирает пути доступа, при помощи которых генерируются полученные данные. Это снимает с плеч программиста огромный груз, повышая таким образом эффективность его работы.

## 6. СКЕПТИЧЕСКОЕ ОТНОШЕНИЕ К РЕЛЯЦИОННЫМ СИСТЕМАМ

Не было недостатка в скептических замечаниях, касающихся целесообразности реляционного подхода к управлению базами данных. Непонимание было источником большей их части, источником других был страх перед многочисленными теоретическими исследованиями, основанными на реляционной модели [1, 2, 15, 16, 24]. Вместо того, чтобы приветствовать создание теоретических основ, обеспечивающих глубину проработки, позиция большинства, кажется, такова: что хорошо в теории, будет плохо на практике. Почти для всех нереляционных СУБД не существует теоретического обоснования, что и является основной причиной того, что их можно назвать *ungepotchket* (это слово из языка идиш, одним из значений которого является «составленный из лоскутков или обрезков»).

С другой стороны, кажутся разумными следующие два вопроса:

(1) Может ли реляционная система обеспечить такой же сервис, какой мы привыкли ожидать от других СУБД?

(2) Если да, то может ли такая система работать не менее эффективно, чем нереляционная СУБД?<sup>1)</sup>

Мы рассмотрим по очереди каждый из этих вопросов.

### 6.1. УРОВЕНЬ СЕРВИСА

Полноценная СУБД предоставляет следующие возможности:

- \* хранение, поиск и обновление данных;
- \* доступный пользователю каталог для описания информации;
- \* поддержка транзакций, чтобы иметь уверенность в том,

---

<sup>1)</sup> Следует помнить, что в нереляционных системах для прикладного программирования всегда используются подязыки описания данных более низкого уровня, чем в реляционных.

что если в базу данных вводится последовательность изменений, то все они или ни одно из них правильно отражены в текущем состоянии базы данных;

- \* возможности восстановления в случае ошибки или отказа (системы, носителей информации или программы);

- \* средства управления параллельным выполнением операций, для того, чтобы одновременная обработка транзакций давала тот же результат, что и последовательное их выполнение в некотором порядке;

- \* поддержка средств контроля доступа (санкционирования доступа, проверки полномочий), чтобы обеспечить соответствие доступа и обработки информации специфическим ограничениям, наложенным на пользователя или программу;

- \* средства объединения (интеграции) и поддержки обмена информацией;

- \* средства обеспечения целостности, чтобы обеспечить соответствие состояния базы данных и изменений этого состояния определенным правилам.

Немногочисленные прототипы реляционных СУБД, созданные в начале 70-х годов, были очень далеки от предоставления всех перечисленных возможностей (вероятно, по очень уважительным причинам). Однако сейчас некоторые реляционные системы, имеющиеся на рынке программного обеспечения, обладают всеми этими возможностями, за исключением последней. Современные версии этих систем, по общему признанию, не обеспечивают в достаточной мере целостность информации, но эта ситуация быстро исправляется [10].

В настоящее время некоторые реляционные СУБД предоставляют более полный набор услуг по обработке данных, чем нереляционные системы. Приведем три примера.

Во-первых, реляционная СУБД обеспечивает выделение всех значимых отношений, существующих в базе данных, в то время как нереляционные системы находят информацию только в тех случаях, когда существуют статически предопределенные пути доступа.

Во-вторых, в качестве примера дополнительных услуг, предоставляемых реляционными системами, рассмотрим разрезы данных (views). *Разрез данных* — это виртуальное отношение (таблица), которая является результатом обработки выражения или последовательности команд. Хотя разрез данных на самом деле не поддерживается непосредственно в текущем состоянии базы данных, он может быть предъявлен пользователю, как если бы он в настоящий момент существовал в базе данных как дополнительная базовая таблица, состояние которой согласовано с состоянием остальных базовых таблиц. Разрезы полезны, так как они позволяют прикладным программистам и пользовате-

лям, сидящим за терминалами, взаимодействовать с постоянными структурами разрезов даже в то время, когда сами базовые таблицы подвергаются структурным изменениям на логическом уровне (при условии, что соответствующие разрезы определены на основе обновленных базовых таблиц). Они также полезны для организации ограничения уровня доступа программ и пользователей. Нереляционные системы или совсем не поддерживают работу с разрезами данных, либо поддерживают гораздо более примитивные эквиваленты, как, например, подсистема в СУБД CODASYL.

И в-третьих, некоторые системы (например, SQL/DS [28] и ее более ранний прототип System R) позволяют делать различные изменения в логической и физической структуре данных в динамическом режиме — одновременно с обработкой транзакций. Обычно внесение таких изменений требует переделки прикладных программ. Таким образом, это свойство уменьшает количество работы, приходящееся на поддержание работоспособности программ, тем самым повышая эффективность труда программистов и давая им возможность заниматься усовершенствованием своих программ, а не их поддержкой. Эту возможность удалось реализовать в системе SQL/DS благодаря тому, что система сама полностью контролирует выбор путей доступа.

В нереляционных системах для внесения таких изменений обычно требуется остановка всей остальной деятельности СУБД, в том числе должно быть прервано и выполнение обрабатываемых транзакций. База данных остается блокированной до тех пор, пока не закончится выполнение организационных изменений, и не будет выполнена компиляция.

## 6.2. ЭФФЕКТИВНОСТЬ

Конечно, люди не будут использовать реляционные системы, если эти системы будут работать медленно. Но все же слишком часто делают ошибочные выводы об эффективности реляционных систем, сравнивая время, которое могло бы потребоваться одной из таких систем для обработки сложной транзакции, с временем, которое потребовалось бы нереляционной системе для выполнения очень простой транзакции. Для того чтобы произвести честное сравнение, нужно сравнивать эти системы на одних и тех же задачах или применениях. Мы представим аргументы, которые покажут, почему реляционные системы должны быть способны соревноваться на равных с нереляционными системами.

Хорошая эффективность определяется двумя факторами: (1) система должна поддерживать физические структуры дан-

ных, ориентированные на увеличение эффективности; (2) запросы к данным, написанные на языке высокого уровня, должны компилироваться в коды более низкого уровня с такой же эффективностью, как это может сделать вручную средний прикладной программист.

Первым аргументом в нашем споре будет тот факт, что программу, написанную на языке того же уровня, что и Кобол, можно эффективно исполнять в больших базах данных, обладающих средствами для вывода данных, структурированных в виде таблиц без видимых пользователю навигационных связей между ними. В поддержку этого аргумента приведем следующую информацию: к августу 1981 г. корпорацией Tandem Computer Corp. было инсталлировано 760 систем, в более чем 700 из них для поддержки баз данных, содержащих сведения о продукции, использовалась реляционная система управления базой данных Tandem ENCOMPASS. Корпорация Tandem доверила свою собственную базу данных, содержащую сведения о производстве, заботам системы ENCOMPASS. Система ENCOMPASS не поддерживает ни видимые пользователю (навигационные) связи, ни невидимые для пользователя связи (пути доступа).

В качестве второго аргумента в споре произведем следующие действия. Возьмем прикладные программы, работающие с описанной выше инсталлированной системой, и запишем все операции поиска и обработки данных с помощью операторов подязыка описания данных, обладающего возможностями реляционной обработки (т. е. SQL). Ясно, что при использовании языка высокого уровня такого типа можно добиться высокой эффективности, только если он компилируется в объектный код (программу на языке транслятора), а не интерпретируется, и этот объектный код достаточно эффективен.

Компиляция использовалась в системе System R и ее коммерческой версии SQL/DS. В 1976 г. Раймонд Лорие придумал остроумную схему пред- и посткомпиляции, чтобы работать с динамическими изменениями в путях доступа. С ее помощью также на ранних этапах (и поэтому эффективно) определяются ограничения на доступ и проверяется целостность (последнее, однако, еще не реализовано). При использовании этой схемы компиляция выражений на подязыке SQL, включенных в прикладную программу, написанную на базовом языке, происходит довольно своеобразно. Шаг компиляции преобразует выражение на языке SQL в соответствующие вызовы (CALLs) в исходной программе и в модули доступа, содержащие объектный код. Эти модули хранятся в базе данных, чтобы потом использовать их в процессе прохождения программы. Код, содержащийся в модулях доступа, генерируется системой так, чтобы



оптимизировать последовательность выполнения основных операций и обеспечить эффективную работу программы. После выполнения такой предкомпиляции прикладная программа компилируется обычным компилятором базового языка. Если впоследствии один или несколько путей доступа будут удалены и будет сделана попытка запустить программу, в модуле доступа останется достаточно информации для того, чтобы система могла скомпилировать новый модуль доступа, содержащий существующие на текущий момент пути доступа, и для этого *не потребует-ся снова компилировать прикладную программу*.

Кстати, тот же самый компилятор с подязыка описания данных используется для обработки непредвиденных запросов (ad hoc queries), поступающих интерактивно с терминала, а также запросов, динамически формируемых в процессе исполнения программы (т.е. с помощью интерактивно введенных параметров). Такие запросы выполняются сразу после компиляции, и, за исключением простейших из них, эффективность выше, чем при использовании интерпретатора.

Генерация модулей доступа (как на стадиях начальной компиляции, так и рекомпиляции) влечет за собой применение довольно сложной схемы оптимизации [27], в которой используются накопленные системой статистические данные, которые в нормальной ситуации неизвестны программисту. Таким образом, средний прикладной программист мог бы соревноваться с этим оптимизатором в создании эффективного кода только для простейшей из транзакций. Любые попытки соревноваться связаны с уменьшением эффективности работы программиста. Таким образом, цена, которую приходится платить за издержки, связанные с увеличением времени компиляции, оказывается вполне приемлемой.

Предполагая, что в обоих случаях используются несвязанные табличные структуры, можно ожидать, что система SQL/DS будет генерировать код, сравнимый по эффективности с кодом, написанным вручную программистом средней квалификации, и превосходящий его по эффективности в сложных случаях. Многие транзакции, используемые в коммерции, крайне просты. Например, может потребоваться найти запись об определенном железнодорожном вагоне, чтобы узнать, где он находится, или подсчитать чьи-либо сбережения. Если поддерживаются достаточно быстрые пути доступа (например, хэшинг), непонятно, почему такие языки высокого уровня, как SQL, QUEL и QBE, должны вырабатывать для этих простых транзакций менее эффективный код, чем язык более низкого уровня, даже если при обработке этих транзакций не используются оптимизирующие возможности компилятора языка высокого уровня.

## 7. НАПРАВЛЕНИЯ ДАЛЬНЕЙШЕЙ РАБОТЫ

Если мы собираемся использовать реляционную базу данных для повышения эффективности, нам необходимо знать, какие усовершенствования можно внести в реляционную систему.

Сначала давайте поговорим об изменениях, осуществимых в ближайшее время. В некоторых реляционных системах необходима более сильная поддержка доменов и первичных ключей, как это предлагается в [10]. Уже было отмечено, что все реляционные системы нуждаются в наращивании вычислительных возможностей, чтобы ограничения, связанные с обеспечением целостности, удовлетворялись автоматически. Существующие ограничения на обновление разрезов данных, полученных в результате операции соединения над отношениями, должны стать менее строгими (где это теоретически возможно), и в решении этой проблемы достигнут значительный прогресс [20]. Необходимо также создавать средства поддержки операции соединения над внешними отношениями.

Методы оптимизации значительно совершенствуются, так что есть основания ожидать дальнейшего увеличения эффективности. В некоторых коммерческих системах, таких как ICL CAFS [22] и в системе IDM500 компании Britton-Lee [13], были созданы специальные средства аппаратной поддержки. В некоторых приложениях эффективность можно увеличивать за счет использования специальных технических средств. Однако в большинстве применений, использующих базы данных, реализованные в виде программ, реляционные базы данных могут соревноваться в эффективности с реализованными в виде программ нереляционными системами.

В настоящее время большинство реляционных систем не обеспечивают специальных средств для работы с инженерными и научными базами данных. Однако необходимость в создании таких средств, в том числе интерфейса с Фортраном, очевидна.

Каталоги в реляционных системах уже состоят из дополнительных отношений, и их можно опрашивать так же, как и остальную информацию в базе данных, используя тот же язык запросов. Естественным усовершенствованием, которое может и должно быть сделано в ближайшее время, является превращение этих каталогов в полностью оснащенные активные управляющие словари для дополнительного текущего контроля информации.

Наконец, в настоящее время можно ожидать появления средств создания баз данных, совместимых с реляционными системами как на физическом, так и на логическом уровнях.

Через более продолжительное время можно предсказать появление баз данных, распределенных по вычислительным сетям

[25, 30, 32] и управляемых таким образом, чтобы прикладные программы и пользователи, работающие в интерактивном режиме, могли работать с информацией так, что (1) как будто она вся хранится в локальном узле — это свойство называется *прозрачностью расположения* (location transparency) (независимость от местоположения, делающая его незаметным для пользователя); и (2) как будто информация нигде не дублируется — это свойство называется *прозрачностью дублирования* (replication transparency) (независимость от дублирования, делающая его незаметным для пользователя).

Все три перечисленных выше проекта основываются на реляционной модели. Свойство реляционности является важным, потому что реляционная база данных обладает большой гибкостью при декомпозиции и распределении ее модулей по сети вычислительных систем и широкими возможностями для динамического объединения распределенной информации. В противоположность этому, базы данных CODASYL DBTG очень трудно разделить на составные части и объединить вновь из-за путаницы навигационных связей между владельцами и членами множеств. Из-за этого подход, реализованный в базе данных CODASYL, очень трудно приспособить к работе в распределенной среде, и это может послужить причиной того, что от него придется отказаться. Вторая причина использования реляционной модели при работе в сети вычислительных систем заключается в том, что для такой модели созданы краткие подязыки описания данных, удобные для передачи запросов между узлами сети.

Можно ожидать, что продолжающаяся работа по развитию возможностей реляционной модели лучше выражать смысл (значение) информации на формальном языке приведет к включению этого смысла в каталог базы данных, чтобы удалить его за рамки прикладных программ и тем самым сделать эти программы еще короче и проще. Здесь, конечно, мы говорим о смысле, который представляется таким образом, что система может его понимать и производить действия над ним.

Развиваются теоретические исследования, касающиеся обработки недостающей или неприменимой информации (как, например, в работе [3]). Результатом этих исследований будут улучшенные методы обработки неопределенных величин.

В настоящее время реляционные базы данных лучше приспособлены для работы с информацией с регулярной или однородной структурой. Сохранятся ли преимущества реляционного подхода при работе с неоднородными данными? Такие данные могут содержать изображения, тексты и разнообразные факты. Ожидается, что на этот вопрос будет дан положительный ответ,

исследования на эту тему развиваются, но их, конечно, еще недостаточно.

Серьезные исследования необходимы для сближения и достижения совместимости языков баз данных и языков программирования. Хорошим примером работы в этом направлении является Pascal/R [26]. Целью продолжающихся исследований является, с одной стороны, включение абстрактных типов данных в языки баз данных [12], и с другой стороны, создание средств реляционной обработки в языках программирования.

## 8. ВЫВОДЫ

Мы представили множество аргументов в поддержку мнения, что технология реляционных баз данных предоставляет возможности для впечатляющего увеличения эффективности как конечным пользователям, так и прикладным программистам. Основными доводами являются такие свойства реляционных систем, как независимость данных, структурная простота и возможности реляционной обработки. Эти свойства были определены для реляционной модели данных и реализованы в виде реляционных систем управления базами данных. Все три перечисленных свойства упрощают создание прикладных программ и формулировку запросов и изменений информации, производимых с терминала. Кроме того, первое свойство делает программы устойчивыми к изменениям описаний и структур в базе данных и тем самым уменьшает трудозатраты на поддержание работоспособности программ.

Почему же тогда в названии лекции утверждается, что реляционная база данных только дает основания для повышения эффективности, а не решает эту задачу полностью? Ответ простой: реляционные базы данных имеют дело только с проблемами взаимодействия с пользователями и с теми компонентами прикладных программ, которые касаются совместно используемых данных. Существует множество других технологий, которые относятся к другим компонентам и аспектам создания баз данных, например, разработка языков программирования, в которых поддерживаются реляционная обработка и методы контроля типов данных, создание редакторов, «понимающих» используемый язык программирования и т. д. Мы использовали слово «основания», потому что взаимодействие с распределенной информацией (на уровне программы или через терминал) является основой большей части деятельности по обработке данных.

Практичность реляционного подхода доказана многочисленными тестами и инсталляцией коммерческих систем, которая уже происходит. Поэтому по мере развития реляционных систем мы можем ожидать резкого увеличения эффективности, которое,

как мы надеялись, появление СУБД обеспечит в первую очередь.

### БЛАГОДАРНОСТИ

Я хотел бы выразить свою признательность членам группы по разработке системы System R из исследовательской лаборатории фирмы IBM в Сан-Хосе за то, что они создали полноценный прототип базы данных, обладающей свойством универсальной реляционности, что привело к изобретению многочисленных нововведений, касающихся языка и организации системы. Я благодарен также разработчикам из лаборатории фирмы IBM в Эндикотте, штат Нью-Йорк, за профессионализм, с которым они превратили систему System R в коммерческий продукт, а также многочисленным сотрудникам университетов, создателям аппаратных средств, фирмам, создающим программное обеспечение и пользователям, которые придумали и реализовали работающие реляционные системы. Я благодарен группе QBE из лаборатории IBM Yorktown Heights, штат Нью-Йорк, группе PRTV научного центра фирмы IBM в Англии и многочисленным ученым, занимавшимся теорией баз данных, для которых реляционная модель была краеугольным камнем. Специальную благодарность я приношу тем немногочисленным коллегам, которые увидели что-то стоящее в моих исследованиях и поддерживали меня на ранних этапах работы, особенно Крису Дейту и Шарон Вейнберг. В конце концов именно Шарон Вейнберг придумала тему этой статьи.

### ЛИТЕРАТУРА

1. Berri C., Bernstein P., Goodman N. A sophisticate's introduction to database normalization theory. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept. 1978.
2. Bernstein P. A., Goodman N., Lai M.-Y. Laying phantoms to rest. Report TR-03-81, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., 1981.
3. Biscup J. A. A formal approach to null values in database relations. *Proc. Workshop on Formal Bases for Data Bases*, Toulouse, France, Dec. 1979; опубликовано в [16] (см. ниже), pp. 299—342.
4. Brodie M., Schmidt J. (Eds.) Report of the ANSI Relational Task Group.
5. Chamberlin D. D. et al. SEQUEL2: A unified approach to data definition, manipulation, and control. *IBM J. Res. & Dev.* 20, 6 (Nov. 1976), 560—565.
6. Chamberlin D. D. et al. A history and evaluation of system R. *Comm. ACM* 24, 10 (Oct. 1981), 632—646.
7. Codd E. F. A relational model of data for large shared data banks. *Comm. ACM* 13, 6 (June 1970), 377—387.
8. Codd E. F. Extending the database relational model to capture more meaning. *ACM TODS* 4, 4 (Dec. 1979), 397—434.
9. Codd E. F. Data models in database management. *ACM SIGMOD Record* 11, 2 (Feb. 1981), 112—114.

10. Codd E. F. The capabilities of relational database management systems. *Proc. Convencio Informatica Llatina*, Barcelona, Spain, June 9—12, 1981, pp. 13—26.
11. Date C. J. Referential integrity. *Proc. Very Large Data Bases*, Cannes, France, Sept. 9—11, 1981, pp. 1—12.
12. Ehrig H., Weber H. Algebraic specification schemes for data base systems. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept. 13—15, 1978, pp. 427—440.
13. Epstein R., Hawthorne P. Design decisions for the intelligent database machine. *Proc. NNC 1980, AFIPS, Vol. 49*, May 1980, pp. 237—241.
14. Eswaran K. P., Chamberlin D. D. Functional specifications of a subsystem for database integrity. *Proc. Very Large Data Bases*, Framingham, Mass., Sept. 1975, pp. 48—68.
15. Fagin R., Horn clauses and database dependencies. *Proc. 1980 ACM SIGACT Symp. on Theory of Computing*, Los Angeles, CA, pp. 123—134.
16. Gallaire H., Minker G., Nicolas J. M. *Advances in Data Base Theory*. Vol. 1, Plenum Press, New York, 1981.
17. Gray J. The transaction concept: Virtues and limitations. *Proc. Very Large Data Bases*, Cannes, France, Sept. 9—11, 1981, pp. 144—154.
18. Griffiths P. G., Wade B. W. An authorisation mechanism for a relational database system. *ACM TODS* 1, 3 (Sept. 1976), 242—255.
19. Held G. ENCOMPASS: A relational data manager. *Data Base/81*, Western Institute of Computer Science, Univ. of Santa Clara, Santa Clara, Calif., Aug. 24—28, 1981.
20. Keller A. M. Updates to relational databases through views involving joins. Report RJ3282, IBM Research Laboratory, San Jose, Calif., October 27, 1981.
21. Lorie R. A., Nilsson J. F. An access specification language for a relational data base system. *IBM J. Res. & Dev.* 23, 3 (May, 1979), 286—298.
22. Maller V. A. J. The content addressable file store—CAFS. *ICL Technical J.* 1, 3 (Nov. 1979), 265—279.
23. Reisner P. Human factors studies of database query languages: A survey and assessment. *ACM Computing Surveys* 13, 1 (March 1981), 13—31.
24. Rissanen J. Theory of relations for databases—A tutorial survey. *Proc. Symp. on Mathematical Foundations of Computer Science*, Zakopane, Poland, September 1978, Lecture Notes in Computer Science, No. 64, Springer Verlag, New York, 1978.
25. Rothnie J. B., Jr., et al. Introduction to a system for distributed databases (SDD-1). *ACM TODS* 5, 1 (March 1980), 1—17.
26. Schmidt J. W. Some high level language constructs for data of type relation. *ACM TODS* 2, 3 (Sept. 1977), 247—261.
27. Selinger P. G., et al. Access path selection in a relational database system. *Proc. 1979 ACM SIGMOD International Conference on Management of Data*, Boston, Mass., May 1979, pp. 23—34.
28. ——— SQL/Data system for VSE; A relational data system for application development. IBM Corp. Data Processing Division, White Plains, N. Y., Feb. 1981.
29. Stonebraker M. R., et al. The design and implementation of INGRES. *ACM TODS* 1, 3 (Sept. 1976), 189—222.
30. Stonebraker M. R., Neuhold E. J. A distributed data base version of INGRES. *Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks*, Lawrence-Berkeley Lab., Berkeley, Calif., May 1977, pp. 19—36.
31. Todd S. J. The Peterlee relational test vehicle — A system overview. *IBM Systems J.* 15, 4 (1976), 285—308.
32. Williams R. et al. R\*: An overview of the architecture. Report RJ 3325, IBM Research Laboratory, San Jose, Calif., Oct. 27, 1981.
33. Zloof M. M. Query by example. *Proc. NCC, AFIPS, Vol. 44*, May 1975, pp. 431—438.

## ПОСТСКРИПТУМ

Э. Ф. Кодд  
Codd and Date Consulting Group  
San Jose, Calif.

Подготавливая доклад по случаю получения премии Тьюринга, я преследовал две цели: (1) подчеркнуть, что реляционная модель определяет не только структурные аспекты организации данных, что хорошо заметно пользователям — она также определяет и способы обработки и аспекты, касающиеся целостности информации; и (2) выделить минимальный набор свойств реляционной модели, которые можно было бы использовать, чтобы отличить реляционные системы управления базами данных (СУБД) от нереляционных систем. Для того, чтобы система могла называться реляционной, необходимо, чтобы она поддерживала каждое из этих свойств.

В первой половине 1980-х годов большинство фирм, продающих программное обеспечение, сообщили о создании коммерческих систем, которые они называли реляционными. Многие из этих фирм считали свои системы «полностью реляционными», в то время как их продукция удовлетворяла только минимальному набору качеств, исходя из которого эти системы можно было назвать реляционными. Некоторые фирмы выпустили на рынок системы, которые не удовлетворяли даже этим минимальным требованиям, но громко именовались в руководствах, рекламных объявлениях, на презентациях и в сообщениях для печати «полностью реляционными».

Чтобы защитить пользователей, которые рассчитывают получить все преимущества, вытекающие из реляционного подхода, осенью 1985 г. я решил опубликовать статью, состоящую из двух частей, «Насколько реляционной является Ваша система управления базой данных?» в журнале *Computerworld* (14 и 21 октября). В первой части этой статьи я описал 12 свойств, каждое из которых коммерческая СУБД должна реализовывать в полной мере, чтобы у нее были шансы действительно считаться реляционной. Во второй части я оценил три широко разрекламированных коммерческих СУБД, рассчитанных на широкое применение. Две из этих систем получили нулевые оценки за реализацию этих 12 свойств.

Я верю, что эти статьи оказали благотворное влияние и уменьшили количество пышных фраз, употребляемых фирмами, продающими реляционные базы данных.

1982

## Обзор вычислительной сложности

Стивен А. Кук

Университет Торонто

Премия Тьюринга 1982 г. была вручена Стивену Артуру Куку, профессору информатики университета Торонто, на ежегодной конференции АСМ в Далласе 25 октября 1982 г. Эта награда — высшее признание Ассоциацией технических достижений в информатике.

Достижения Кука охарактеризованы следующим образом: «Д-р Кук существенно и глубоко продвинул наше понимание сложности вычислений. Его плодотворная работа о сложности процедуры доказательства теорем, представленная на симпозиуме АСМ SIGACT по теории вычислений 1971., заложила основы теории  $NP$ -полноты. Последовавшие за этим исследование границ и природы  $NP$ -полных классов задач было одним из наиболее активных и важных направлений в информатике за последнее десятилетие.

Кук хорошо известен благодаря своим результатам, оказавшим большое влияние на фундаментальные области информатики. Он внес существенный вклад в теорию сложности, соотношения затрат времени и объема памяти при вычислениях и в логику языков программирования. Его работы характеризуются элегантностью и интуицией и высвечивают истинную природу вычислений».

В течение 1970—1979 гг. Кук интенсивно работал при поддержке Национального исследовательского совета. В 1977—1979 гг. он получал стипендию имени Э. У. Р. Стеси. Автор многочисленных основополагающих статей, в настоящее время он занят доказательством того, что не существует «хорошего» алгоритма для  $NP$ -полных задач.

Тьюринговская премия АСМ учреждена в память А. М. Тьюринга, английского математика, сделавшего крупный вклад в информатику.

Представлен исторический обзор сложности вычислений. Акцент сделан на два фундаментальных вопроса — определение внутренней вычислительной сложности задачи и доказательство верхних и нижних оценок сложности задач. Обсуждаются вероятностные и параллельные вычисления.

Это вторая лекция лауреата премии Тьюринга по вычислительной сложности. Первая была прочитана Микаэлем Раби-



ном в 1976 году<sup>1)</sup>. Сейчас при чтении отличной статьи Рабина [62] мое внимание привлекает то обстоятельство, насколько возросла с тех пор активность в данной области. В этом кратком обзоре я хочу упомянуть наиболее важные и интересные на мой взгляд результаты по сложности вычислений с начала этих исследований примерно в 1960 г. В столь обширной области выбор тем неизбежно индивидуален, однако я надеюсь, что включенные в обзор статьи по любым стандартам являются фундаментальными.

## 1. РАННИЕ РАБОТЫ

Предыстория вопроса восходит к Алану Тьюрингу. В своей работе 1937 г. «О вычислимых числах с приложением к проблеме разрешения» [85] Тьюринг ввел свою знаменитую машину, которая обеспечила наиболее убедительную (по тому времени) формализацию понятия эффективно (или алгоритмически) вычислимой функции. Как только это понятие было точно сформулировано, стали возможными доказательства невыполнимости вычислений для компьютеров. В той же статье Тьюринг доказал, что никакой алгоритм (т.е. машина Тьюринга) не может по произвольной формуле исчисления предикатов решить за конечное число шагов, выполнима ли эта формула.

После того как теория, объяснявшая, какие задачи могут и какие не могут быть решены компьютером, стала хорошо разработанной, было естественно задаться вопросом о сравнительной вычислительной сложности вычислимых функций. Этим и занимается теория сложности вычислений. Рабин [59, 60] был одним из первых (1960), кто в явном виде сформулировал следующий вопрос: что означает, что  $f$  труднее вычислить, чем  $g$ ? Рабин предложил аксиоматический подход, заложивший основы абстрактной теории сложности, развитой Блюмом [6] и другими.

Вторая важная ранняя (1965) работа — статья Хартманиса и Стирнса «О вычислительной сложности алгоритмов» [37]<sup>2)</sup>. Эта статья стала широко известна и дала название описываемой области исследований. Было введено важное понятие меры сложности как времени вычислений на многоленточных машинах Тьюринга, и были доказаны теоремы об иерархии. Эта статья также поставила один интригующий вопрос, все еще остающийся открытым. А именно, вычислимо ли за реальное время любое алгебраическое иррациональное число (вроде  $\sqrt{2}$ ), т.е. сущест-

<sup>1)</sup> Микаэль Рабин и Дана Скотт удостоены Тьюринговской премии 1976 г.

<sup>2)</sup> Интересные воспоминания см. у Хартманиса [36].

вует ли машина Тьюринга, печатающая десятичное разложение этого числа со скоростью один разряд за 100 шагов?

Третья основополагающая статья (1965) — «Внутренняя вычислительная трудность функций» Алана Кобэма [15]. Кобэм подчеркивает слово «внутренняя» (intrinsic), т. е. его интересует машинно-независимая теория. Он спрашивает, сложнее ли умножение, чем сложение, и считает, что на этот вопрос невозможно ответить, покада соответствующая теория не будет полностью развита. Кобэм также определил и охарактеризовал важный класс функций, который он обозначил  $\mathcal{L}$ : это те функции натуральных чисел, которые вычислимы за время, ограниченное полиномом от десятичной длины входа.

Три другие статьи, повлиявшие на упомянутых авторов, так же как и на других исследователей теории сложности (включая меня), — статьи Ямады [91], Беннета [4] и Ритчи [66]. Интересно отметить, что Рабин, Стирнс, Беннет и Ритчи — все были студентами в Принстоне примерно в одно и то же время.

## 2. РАННИЕ РЕЗУЛЬТАТЫ И ПОНЯТИЯ

Некоторые из авторов ранних работ задавались вопросом: что же на самом деле следует считать мерой сложности? Большинство выбирало время вычисления или объем памяти как наиболее очевидные характеристики сложности, но не были убеждены, что эти характеристики единственные или правильные. Например, Кобэм [15] отмечал: «...некоторая мера, связанная с физическим понятием работы, возможно, приведет к наиболее удовлетворительному анализу». Рабин [60] ввел аксиомы, которым должна удовлетворять мера сложности. Опираясь на двадцатилетний опыт, я теперь считаю очевидным, что время и объем памяти — особенно время — находятся в ряду наиболее существенных мер сложности. Представляется, что при оценке эффективности алгоритма в первую очередь принимается в расчет время его выполнения. Однако в последнее время становится ясным, что параллельное время и размеры оборудования — также важные меры сложности (см. разд. 6).

Другая важная мера сложности, восходящая в каком-то смысле еще к Шеннону [74] (1949), — сложность булевой схемы (или комбинационная). Здесь удобно предположить, что рассматриваемая функция  $f$  преобразует конечные цепочки бит в конечные цепочки бит, и сложность  $C(n)$  функции  $f$  — размер наименьшей булевой схемы, вычисляющей  $f$  для всех входных наборов длины  $n$ . Эта очень естественная мера тесно связана с временем вычисления (см. [57 a], [57 b], [68 b]) и имеет свою хорошо развитую теорию (см. Савидж [68 a]).

Другой вопрос, поставленный Кобэмом [15], состоит в том;

что же представляет собой «шаг» в вычислении. Это равносильно вопросу, что именно является правильной моделью компьютера для измерения времени работы алгоритма. Обычно в литературе используют многоленточные машины Тьюринга, но им присущи искусственные ограничения с точки зрения эффективной реализации алгоритмов. Например, не существует разумного объяснения, почему память должна быть организована линейно. Почему бы не плоские массивы деревьев? Почему бы не допустить память с произвольным доступом?

Фактически с 1960 г. было предложено довольно много моделей компьютеров. Поскольку в реальных компьютерах память с произвольным доступом (RAM) имеется, представляется естественным допустить ее и в модели. Но как именно сделать это — вопрос не очевидный. Если машина может запомнить целые числа за один шаг, должны быть введены ограничения на их величину. (Если число 2 возвести 100 раз в квадрат, то результат имеет  $2^{100}$  бит, которые не уместятся во все существующие средства памяти.) В [19] я предложил платную RAM, в которой плата (число шагов) размером примерно  $\log|x|$  взимается всякий раз, когда  $x$  запоминается или извлекается из памяти. Этот подход работает, но не является вполне убедительным. Более популярная модель произвольного доступа та, что использовалась Ахо, Хопкрофтом и Ульманом [3], в которой каждая операция над целым числом имеет единичную стоимость, но не допускается, чтобы целые числа становились не разумно большими (например, их величина может быть ограничена некоторым фиксированным полиномом, зависящим от размера входа). Вероятно, с математической точки зрения наиболее удовлетворительной является машина с модификацией памяти Шёнхаге [69], которую можно рассматривать либо как машину Тьюринга, строящую свою собственную структуру памяти, либо RAM с единичной стоимостью, которая за один шаг может только копировать, прибавлять (или вычитать) единицу или запоминать (или извлекать из памяти). Машина Шёнхаге — это слегка обобщенная машина Колмогорова — Успенского [46], предложенная гораздо раньше (1958), и мне кажется, что это наиболее общая машина, которая за один шаг выполняет ограниченную работу. Беда в том, что она, вероятно, слишком мощная. (См. разд. 3 «Умножение больших чисел».)

Возвращаясь к вопросу Кобэма «что есть шаг», я думаю, что за последние 20 лет стало ясным, что единого четкого ответа на этот вопрос нет. К счастью, конкурирующие компьютерные модели не слишком сильно отличаются временем вычислений. Вообще говоря, каждая из них может моделировать любую другую самое большее с квадратичным увеличением объема вычислений (некоторые из первых аргументов в пользу этого

приведены в [37]). Что касается широко распространенных (основных) моделей произвольного доступа, то они отличаются только множителем, логарифмически зависящим от времени вычислений.

Это приводит к последнему важному понятию, выработанному к 1965 г., — идентификации класса задач, разрешимых за время, ограниченное полиномом от длины входа. Различие между полиномиальными и экспоненциальными алгоритмами было проведено еще в 1953 г. фон Нейманом [90]. Однако этот класс не был определен формально и не изучался, пока Кобэм [15] не ввел в 1964 г. класс  $\mathcal{P}$  функций. Кобэм указал, что этот класс определен корректно независимо от того, какая компьютерная модель выбрана, и охарактеризовал его в духе теории рекурсивных функций. Мысль о том, что вычислимость за полиномиальное время, грубо говоря, соответствует практической выполнимости (tractability), была впервые высказана в печати Эдмондсом [27], который назвал полиномиальные алгоритмы «хорошими алгоритмами». Принятое теперь стандартное обозначение  $P$  для класса множеств цепочек, распознаваемых за полиномиальное время, было позднее введено Карпом [42].

Отождествление  $P$  с практически выполнимыми (или осуществимыми) задачами стало общепринятым с начала 1970-х. Не является непосредственно очевидным, почему это должно быть истинным, так как алгоритм, время выполнения которого есть полином  $n^{1000}$ , конечно, неосуществим, и наоборот, алгоритм, время выполнения которого экспоненциально зависит от  $n$  по формуле  $2^{0,00001n}$ , практически осуществим. Представляет эмпирическим фактом, однако, что для естественно возникающих задач нет оптимальных алгоритмов с такими временами работы<sup>1)</sup>. Самый известный практический алгоритм с экспоненциальным временем работы в худшем случае — симплексный алгоритм линейного программирования. Смэйл [75, 76] попытался объяснить это, показав, что в некотором смысле среднее время работы невелико, но также важно отметить, что Хачиян [43] показал принадлежность линейного программирования классу  $P$ , воспользовавшись другим алгоритмом. Таким образом, наш общий тезис, что класс  $P$  эквивалентен классу практически осуществимых задач, не нарушен.

### 3. ВЕРХНИЕ ОЦЕНКИ ВРЕМЕНИ

Значительная доля исследований в информатике состоит из построения и анализа огромного числа эффективных алгорит-

<sup>1)</sup> См. [31], с. 6—9, где это обсуждается.

мов. Наиболее важные алгоритмы (с точки зрения вычислительной сложности) должны быть в некотором смысле особенными; они в основном дают удивительно быстрый способ решения какой-нибудь простой или важной задачи. Ниже я перечисляю некоторые из наиболее интересных алгоритмов, изобретенных начиная с 1960 г. (Кстати, интересно порассуждать о том, какие именно алгоритмы признаны за это время наиболее важными. Бесспорно, арифметические операции  $+$ ,  $-$ ,  $*$ ,  $\div$  над десятичными числами являются основными. После этого, я думаю, следующими кандидатами являются быстрая сортировка и поиск, исключение по Гауссу, алгоритм Эвклида и симплексный алгоритм.)

Параметр  $n$  относится к размеру входа, а временные оценки суть оценки времени в худшем случае и относятся к многоленточной машине Тьюринга (или любой разумной машине с произвольным доступом к памяти), за исключением случаев, где это будет специально оговариваться.

(1) **Быстрое преобразование Фурье** [23], требующее  $O(n \log n)$  арифметических операций, — один из наиболее часто используемых алгоритмов в научных вычислениях.

(2) **Умножение больших чисел.** Школьный метод требует  $O(n^2)$  битовых операций для умножения двух  $n$ -разрядных чисел. В 1962 г. Карацуба и Офман [41] опубликовали метод, требующий всего  $O(n^{1.59})$  шагов. Вскоре после этого Тоом [84] показал, как построить булеву схему сложности  $O(n^{1+\epsilon})$  для произвольно малого  $\epsilon > 0$ , выполняющую умножение. Я был в это время студентом старшего курса в Гарварде, и, вдохновленный вопросом Кобэма: «Сложнее ли умножение, чем сложение?», наивно пытался доказать, что умножение требует  $\Omega(n^2)$  шагов на многоленточной машине Тьюринга. Статья Тоома сильно удивила меня. С помощью Стола Аандераа [22] я показал, что умножение требует  $\Omega(n \log n / (\log \log n)^2)$  шагов машины Тьюринга<sup>1)</sup> при вычислении «on line». Я отметил в своей диссертации, что метод Тоома может быть приспособлен к многоленточным машинам Тьюринга, с тем чтобы выполнить умножение за  $O(n^{1+\epsilon})$  шагов, — то, что, я уверен, Тоома не удивило бы.

В настоящий момент асимптотически наилучшее время выполнения умножения чисел на многоленточной машине Тьюринга есть  $O(n \log n \log \log n)$  — результат, полученный Шёнхаге и Штрассеном [70] (1971) с использованием быстрого преобразования Фурье. Однако Шёнхаге [69] недавно показал посредством сложного рассуждения, что его машины с модификацией памяти (см. разд. 2) могут умножать за  $O(n)$  (линейное

<sup>1)</sup> Эта нижняя оценка была слегка улучшена. См. [56] и [64].

время!). Мы вынуждены заключить, что либо умножение проще, чем мы думали, либо машины Шёнхаге «жуютничают».

(3) **Умножение матриц.** Очевидный метод требует  $n^2(2n-1)$  арифметических операций для умножения двух матриц размера  $n \times n$ , и в 1950—60-е годы пытались доказать, что этот метод оптимальный. Неожиданно Штрассен [81] (1969) опубликовал свой метод, требующий только  $4,7n^{2,81}$  операций. Серьезные усилия были приложены к уменьшению показателя  $2,81$ , и сейчас наилучшее известное время  $O(n^{2,496})$  операций — результат, принадлежащий Коперсмит и Винограду [24]. Здесь еще большой простор для улучшения оценки, потому что наилучшая известная нижняя оценка  $2n^2-1$  (см. [13]).

(4) **Максимальные паросочетания в общих неориентированных графах.** Возможно, это была первая среди принадлежащих  $P$  задач, для которой было явно показано, что алгоритм, принадлежащий классу  $P$ , трудный. В оказавшей заметное влияние статье Эдмондса [27] излагался результат и обсуждалось понятие алгоритма, выполнимого за полиномиальное время (см. разд. 2). Он также отметил, что простое понятие расширяющегося пути, которого достаточно для двудольного случая, не годится для неориентированных графов общего вида.

(5) **Распознавание простых чисел.** Основной вопрос здесь: принадлежит ли эта задача классу  $P$ ? Другими словами, существует ли алгоритм, который всегда сообщает нам, является ли произвольное  $n$ -разрядное число на входе простым, и останавливается после числа шагов, ограниченных фиксированным полиномом от  $n$ ? Гэри Миллер [53] (1976) показал, что такой алгоритм существует, но его доказательство опирается на расширенную гипотезу Римана. Соловей и Штрассен [77] построили быстрый алгоритм Монте-Карло (см. разд. 5) для распознавания простого числа, но если входное число составное, то с малой вероятностью алгоритм может ошибочно распознать его как простое. Наилучший из доказуемых детерминированных алгоритмов принадлежит Адлеману, Померанцу и Рамли [2], он требует времени  $n^{O(\log \log n)}$ , что чуть хуже полиномиального. Его вариацией является алгоритм Коэна и Ленстры мл. [17], который может, как правило, обрабатывать числа до 100 десятичных разрядов примерно за 45 с.

Недавно показана принадлежность классу  $P$  трех важных задач. Первая — линейное программирование, для которого это было доказано Хачияном [43] в 1979 г. (Изложение можно найти, например, в [55].) Для второй — распознавания изоморфизма двух графов степени не более  $d$  — вопрос решен Лаксом [50] в 1980 г. (Алгоритм полиномиален относительно числа вершин при фиксированном  $d$ , но экспоненциален относительно  $d$ .) Третья — разложение на множители полиномов

с рациональными коэффициентами. Это было показано для полиномов от одной переменной Ленстрой, Ленстрой и Ловасом [48] в 1982 г. Как показывает результат Калтофена [39, 40], это можно обобщить на полином от любого фиксированного числа переменных.

#### 4. НИЖНИЕ ОЦЕНКИ

Самая серьезная проблема в теории сложности, отделяющая эту теорию от анализа алгоритмов, — доказательство нижних оценок сложности отдельных задач. Есть нечто вызывающее большое удовлетворение, когда удается доказать, что задача, требующая ответа да—нет, не может быть решена за  $n$  или  $n^2$  или  $2^n$  шагов независимо от того, какой алгоритм применяется для ее решения. В доказательстве нижних оценок удалось многого достичь, но остались открытыми еще более важные и обескураживающие вопросы.

Все важные нижние оценки времени вычислений и объема памяти основаны на «диагональных рассуждениях». Диагональные рассуждения использовались Тьюрингом и его современниками для доказательства того, что некоторые задачи алгоритмически неразрешимы. Они были также использованы еще до 1960-х годов для определения иерархии вычислимых 0—1 функций<sup>1)</sup>. В 1960 г. Рабин [60] доказал, что для всякой разумной меры сложности, такой, как время вычисления или объем памяти, достаточное увеличение допустимого времени или объема памяти всегда позволяет вычислять большее число 0—1 функций. Примерно в то же время Ритчи в своей диссертации [65] определил специальную иерархию функций (которая, как он показал, нетривиальна для 0—1 функций) в терминах объема доступной памяти. Несколько позднее результат Рабина был отчасти улучшен для требуемого времени на многоленточных машинах Тьюринга Хартманисом и Стирнсом [37] и для объема памяти — Стирнсом, Хартманисом и Льюисом [78].

##### 4.1. ДОКАЗАТЕЛЬСТВО ПРАКТИЧЕСКОЙ НЕОСУЩЕСТВИМОСТИ ЕСТЕСТВЕННЫХ РАЗРЕШИМЫХ ЗАДАЧ

Упомянутые выше результаты по иерархиям дают нижние оценки для времени и памяти, необходимых для вычисления конкретных функций, но все такие функции кажутся «надуманными». Например, легко видеть, что функция  $f(x, y)$ , которая дает первый разряд на выходе машины  $x$  при входе  $y$  после

<sup>1)</sup> См., например, Гжегорчик [35].

$(|x| + |y|)^2$  шагов, не может быть вычислена за время  $(|x| + |y|)^2$ . Только в 1972 г., когда Альберт Мейер и Ларри Стокмейер [52] доказали, что проблема эквивалентности для регулярных выражений с возведением в квадрат требует экспоненциальной памяти и, следовательно, экспоненциального времени, была найдена нетривиальная нижняя оценка для общих моделей вычислений для «естественной» задачи (естественной в смысле интересной и имеющей смысл независимо от вычислительных машин). Вскоре после этого Мейер [51] нашел очень сильную нижнюю оценку времени, требуемого для определения истинности формул в некоторой формально разрешимой теории, именуемой WSIS (слабая монадическая теория следования второго порядка). Он доказал, что никакой компьютер, время работы которого ограничено фиксированным числом экспонент ( $2^n$ ,  $2^{2^n}$ ,  $2^{2^{2^n}}$  и т.д.), не может правильно разрешить WSIS. Аспирант Мейера Стокмейер вычислил [79], что всякая булева схема (считай, компьютер), которая корректно решает задачу об истинности произвольной WSIS формулы длины 616 символов, должна иметь более чем  $10^{123}$  элементов. Число  $10^{123}$  было выбрано как число протонов, которые могут поместиться в известной части Вселенной. Это очень убедительное доказательство практической неосуществимости!

После Мейера и Стокмейера было получено большое число нижних оценок сложности разрешимых формальных теорий (обзоры см. [29, 80]). Одной из наиболее интересных является нижняя оценка в форме двойной экспоненты для времени, требуемого для разрешения пресбургеровской арифметики (теория натуральных чисел со сложением), полученная в работе Фишера и Рабина [30]. Это недалеко от наилучшей известной верхней оценки времени для этой теории, которая представляет собой тройную экспоненту [54]. Наилучшая верхняя оценка для памяти — двойная экспонента [29].

Несмотря на перечисленные успехи, отметим, что список доказанных нижних оценок в задачах меньшей сложности шокирует. Фактически неизвестны нелинейные нижние оценки времени для общего вида вычислительных моделей ни для какой естественной задачи класса  $NP$  (см. разд. 4.4), в частности, ни для одной из 300 проблем, перечисленных в [31]. Конечно, можно доказать из диагональных соображений существование задач класса  $NP$ , требующих времени  $n^k$  для любого фиксированного  $k$ . Однако для нижних оценок памяти мы даже не знаем, как доказывать существование  $NP$ -задач, не решаемых с памятью  $O(\log n)$  для off-line машин Тьюринга (см. разд. 4.3). И это несмотря на то, что наиболее известные верхние оценки объема памяти для многих естественных случаев по существу линейны относительно  $n$ .



## 4.2. СТРУКТУРИРОВАННЫЕ НИЖНИЕ ОЦЕНКИ

Несмотря на то что успехи в доказательстве интересных нижних оценок для конкретных задач на общих вычислительных моделях невелики, для «структурированных» моделей такие результаты есть. Термин «структурированный» был введен Бородиным [9] для компьютеров с ограниченным набором операций, подходящим для рассматриваемой задачи. Простой пример этого — задача сортировки  $n$  чисел. Можно доказать (см. [44]) без особых затруднений, что это требует по крайней мере  $n \log n$  сравнений при условии, что единственная допустимая операция компьютера над выходами — попарное сравнение входов. Эта нижняя оценка ничего не говорит о машинах Тьюринга или булевых схемах, но она была распространена на машины с единичной стоимостью обращения к памяти с произвольным доступом в предположении, что деление запрещено.

Вторая и очень элегантная структурированная нижняя оценка, полученная Штрассеном [82] (1973), устанавливает, что полиномиальная интерполяция, т.е. нахождение коэффициентов полинома степени  $n-1$ , проходящего через  $n$  заданных точек, требует в предположении, что допустимы только арифметические операции  $\Omega(n \log n)$  умножений. Интересно это отчасти потому, что первоначальное доказательство Штрассена основано на теореме Безу, глубококом результате алгебраической геометрии. Совсем недавно Баур и Штрассен [83] усилили нижнюю оценку, показав, что даже средний коэффициент интерполяционного полинома по  $n$  точкам требует для своего вычисления  $\Omega(n \log n)$  умножений.

Привлекательность всех этих структурированных результатов состоит в том, что нижние оценки близки к наилучшим известным верхним оценкам<sup>1)</sup> и наилучшие известные алгоритмы могут быть реализованы на структурированных моделях, к которым относятся нижние оценки. (Отметим, что корневая сортировка, про которую иногда говорят, что она требует линейного времени, на самом деле требует по крайней мере  $n \log n$  шагов, если предположить, что числа на входах имеют достаточное число разрядов, чтобы все они могли быть различными.)

## 4.3. НИЖНИЕ ОЦЕНКИ ПРОИЗВЕДЕНИЯ ВРЕМЕНИ И ПАМЯТИ

Другой путь обойти трудности доказательства «абсолютных» нижних оценок времени и объема памяти состоит в доказательстве нижних оценок времени в предположении малой памяти. Кобэм [16] доказал первый такой результат

<sup>1)</sup> Верхние оценки для интерполяции см. у Бородина и Манро [12].

в 1966 г., когда он показал, что произведение времени и памяти для распознавания  $n$ -разрядного совершенного квадрата на off-line машине Тьюринга должно быть  $\Omega(n^2)$ . (То же самое верно для  $n$ -символьных палиндромов.) Здесь входное слово записано на двухдорожечной входной ленте, с которой возможно только чтение, и используемая память по определению — число квадратов, просмотренных рабочими лентами, имеющимися в данной машине Тьюринга. Так, если, например, память ограничена  $O(\log^3 n)$  (чего более чем достаточно), то время должно быть  $\Omega(n^2/\log^3 n)$  шагов.

Слабость результата Кобэма состоит в том, что, хотя off-line машина Тьюринга разумна для измерения времени вычислений и памяти в отдельности, эта модель слишком ограничительна, когда память и время рассматриваются вместе. Например, ясно, что палиндромы могут быть распознаны за  $2n$  шагов с постоянной зоной, если две головки осуществляют совместный просмотр входной ленты. Бородин и я [10] частично исправили эту слабость, когда мы доказали, что сортировка  $n$  целых чисел по возрастанию в диапазоне от 1 до  $n^2$  требует произведения времени и памяти  $\Omega(n^2/\log n)$ . Доказательство применимо к любой «последовательной машине общего вида», что включает off-line машину Тьюринга со многими входными головками или даже с произвольным доступом к входной ленте. К сожалению, в нашем доказательстве очень важный момент состоит в том, что сортировка требует много выходных битов, и остается открытым вопрос: можно ли подобную нижнюю оценку получить для какой-нибудь задачи распознавания множества, такой, как распознавание того, все ли  $n$  входных чисел различны? (Наша нижняя оценка для сортировки недавно слегка улучшена в [64].)

#### 4.4. NP-ПОЛНОТА

Теория NP-полноты, конечно, самое значительное достижение в теории сложности вычислений. Я не буду здесь на ней останавливаться, потому что это теперь хорошо известно и описано в учебниках. В частности, книга Гэри и Джонсона [31] — прекрасный источник сведений по этому вопросу.

Класс NP состоит из всех множеств, распознаваемых за полиномиальное время на недетерминированной машине Тьюринга. Насколько мне известно, впервые математически эквивалентный класс был определен Джеймсом Беннетом в его докторской диссертации [4] в 1962 г. Беннет использовал для этого класса название «расширенные положительные рудиментарные отношения», и в его определении использованы логические кванторы вместо вычислительных машин. Я прочитал эту часть его дис-

сертации и осознал, что его класс может быть охарактеризован как то, что сейчас называют  $NP$ . Я использовал обозначение  $\mathcal{L}^+$  (вслед за классом  $\mathcal{L}$  Кобэма) в моей статье [18] в 1971 г., а Карп дал ныне принятое название  $NP$  для этого класса в своей статье [42] в 1972 г. Тем временем совершенно независимо Эдмондс еще в 1965 г. [28] рассуждал неформально о задачах с «хорошими характеристиками» — понятии, по существу эквивалентном  $NP$ .

В 1971 г. [18] я ввел понятие  $NP$ -полноты и доказал, что 3-выполнимость и задача о подграфе  $NP$ -полны. Год спустя Карп [42] доказал  $NP$ -полноту 21 задачи, продемонстрировав тем самым важность этого предмета. Независимо от этого и чуть-чуть позднее Леонид Левин [49] в Советском Союзе (теперь в Бостонском университете) определил подобное (и более сильное) понятие и доказал про шесть задач, что они полны в этом смысле. Неформальное понятие «переборной задачи» было стандартным в советской литературе, и Левин назвал свои задачи «универсальными переборными задачами».

Класс  $NP$  включает огромное количество практических задач, появляющихся в бизнесе и промышленности (см. [31]). Доказательство того, что  $NP$ -задача  $NP$ -полна, есть доказательство того, что задача не принадлежит классу  $P$  (не имеет детерминированного алгоритма с полиномиальным временем), разве что всякая  $NP$ -задача принадлежит классу  $P$ . Поскольку последнее условие революционизировало бы информатику практический результат  $NP$ -полноты — это нижняя оценка. Это и явилось причиной того, что я включил этот предмет в раздел о нижних оценках.

#### 4.5. $\#P$ -ПОЛНОТА

Понятие  $NP$ -полноты применимо к множествам, и доказательство того, что множество  $NP$ -полно, обычно интерпретируется как доказательство его необозримости. Есть, однако, большое количество очевидно невычислимых функций, для которых, по-видимому, нельзя найти никакого доказательства  $NP$ -полноты, к ним применимого. Лесли Валиант [86, 87], чтобы выйти из положения, ввел понятие  $\#P$ -полноты. Доказательство того, что функция  $\#P$ -полна, показывает, что она практически не поддается вычислению, тем же самым способом, что и доказательство  $NP$ -полноты множества показывает, что оно практически необозримо для распознавания; а именно, если  $\#P$ -полная функция вычислима за полиномиальное время, то  $P = NP$ .

Валиант дал много примеров  $\#P$ -полноты функций, но, возможно, наиболее интересный из них — перманент целочисленной матрицы. Перманент определяется формально очень похоже на

детерминант, но в то время как детерминант легко вычисляется посредством гауссова метода исключения, все многочисленные попытки за последние сто с лишним лет найти подходящий метод вычисления перманента были безуспешными. Валиант дал первое убедительное объяснение этой безуспешности, когда доказал, что перманент  $\#P$ -полон.

## 5. ВЕРОЯТНОСТНЫЕ АЛГОРИТМЫ

Использование случайных чисел для моделирования или аппроксимации случайных процессов вполне естественно и стало общепринятым в вычислительной практике. Однако идея о том, что случайные входы могут быть полезны в решении детерминированных комбинаторных задач, гораздо медленнее проникала в сообщество специалистов по информатике. Здесь я ограничусь вероятностными (бросание монеты) алгоритмами с полиномиальным временем, которые «решают» (в некотором разумном смысле) задачу, для которой детерминированные алгоритмы с полиномиальным временем неизвестны.

Первым таким алгоритмом, кажется, был алгоритм Берлекэмпа [5], 1970 г., для разложения на множители полинома  $f$  над полем  $GF(p)$  с  $p$  элементами. Алгоритм Берлекэмпа работает полиномиальное время от степени  $f$  и  $\log p$  и с вероятностью по крайней мере  $1/2$  находит неприводимые разложения на сомножители  $f$ ; в противном случае его работа заканчивается неудачей. Так как алгоритм можно повторять любое число раз и неудачные исходы все независимы, на практике алгоритм всегда разлагает полином за приемлемое (разумное) время.

Более сильный пример — алгоритм распознавания простых чисел Соловея и Штрассена [77], предложенный в 1974 г. Этот алгоритм работает полиномиальное время от длины входа  $m$  и выдает результат «простое» или «составное». Если  $m$  и в самом деле простое, то на выходе определено «простое», но если  $m$  составное, то с вероятностью самое большее половина ответ может также оказаться «простое». Алгоритм с  $m$  на входе можно повторять любое число раз, и все результаты будут независимы от предшествующих попыток. Значит, если ответ всегда «составное», то пользователь знает, что  $m$  составное: если результат «простое» неизменно выдается после, скажем, 100 испытаний, то пользователь имеет веские основания считать, что  $m$  простое, так как всякое фиксированное составное  $m$  давало бы такой результат с очень малой вероятностью (меньше чем  $2^{-100}$ ).

Рабин [61] разработал еще один вероятностный алгоритм со свойствами, подобными вышеупомянутому, и нашел его очень быстрым при пробах на компьютере. Число  $2^{400} - 593$  было рас-

познано как простое (с высокой вероятностью) за несколько минут.

Одно из интересных приложений вероятностных проверок чисел на простоту было предложено Райвестом, Шамиром и Адлеманом [67a] в их основополагающей статье по криптосистемам общего доступа в 1978 г. Их система требует порождения больших (100-значных) случайных простых чисел. Они предложили проверять случайные 100-значные числа, используя метод Соловея—Штрассена, откуда этот метод не обнаружит «вероятностно» простое число в описанном выше смысле. На самом деле, с помощью нового мощного детерминированного метода проверки чисел на простоту Коэна и Ленстры [17], упомянутого в разд. 3, после обнаружения случайного 100-значного «вероятностного простого числа» его можно проверить примерно за 45 с., если важно знать наверняка.

Класс множеств, распознаваемых вероятностными алгоритмами с полиномиальным временем работы в смысле Соловея и Штрассена, известен в литературе как  $R$  (или иногда как  $RP$ ). Таким образом, множество принадлежит классу  $R$  тогда и только тогда, когда для него существует вероятностный алгоритм распознавания, срабатывающий всегда за полиномиальное время и никогда не ошибающийся, если вход не принадлежит  $R$ , и для каждого входа из  $R$  на его выходе правильный ответ для каждого прогона алгоритма появляется с вероятностью по крайней мере  $1/2$ . Следовательно, множество составных чисел принадлежит  $R$ , и вообще  $P \subseteq R \subseteq NP$ . Существуют другие интересные примеры множеств из  $R$ , про которые неизвестно, принадлежат ли они классу  $P$ . Например, Шварц [71] показал, что множество невырожденных матриц, элементы которых — полиномы от многих переменных, принадлежит  $R$ . Алгоритм вычисляет полиномы для случайных небольших целочисленных значений переменных и вычисляет детерминант результата. (Детерминант, по-видимому, не может быть обозримым образом вычислен непосредственно, потому что вычисляемые полиномы имели бы в общем случае экспоненциально много членов.)

Любопытен открытый пока вопрос: верно ли, что  $R = P$ ? Заманчиво ответить на него «да», исходя из философских соображений, что случайное бросание монеты не может принести много пользы, когда ответ должен быть определенным — да или нет. С ним связан другой вопрос: является ли вероятностный алгоритм (показывающий принадлежность задачи классу  $R$ ) для всех практических целей столь же хорошим, что и детерминированный алгоритм? В конце концов, вероятностные алгоритмы можно выполнять, используя генераторы псевдослучайных чисел, имеющиеся для большинства компьютеров, и вероятность ошибки  $2^{-100}$  пренебрежимо мала. Загвоздка в том, что

генераторы псевдослучайных чисел не генерируют на самом деле случайные числа, и непонятно, насколько хорошо они будут работать для данного вероятностного алгоритма. Практический опыт показывает, что они работают, видимо, хорошо. Но если они *всегда* работают хорошо, то, следовательно,  $R=P$ , потому что псевдослучайные числа порождаются детерминированно, так что подлинная случайность в конце концов не нужна. Другая возможность состоит в использовании физического процесса, такого, как тепловой шум, для порождения случайных чисел. Но насколько реальной может быть случайность в природе — это открытый вопрос философии науки.

Позвольте мне завершить этот раздел упоминанием интересной теоремы Адлемана [1] о классе  $R$ . Легко видеть [57b], что если множество принадлежит классу  $P$ , то для каждого  $n$  существует булева схема размера, ограниченного полиномом от  $n$ , которая определяет, всегда ли произвольная последовательность символов длины  $n$  принадлежит этому множеству. Адлеман доказал, что то же самое истинно для класса  $R$ . Так, например, для каждого  $n$  существует малая «схема вычисления» (компьютерная схема), которая корректно и быстро распознает простоту  $n$ -разрядных чисел. Загвоздка в том, что схемы не регулярны по  $n$ , и, возможно, построение схемы для 100-разрядных чисел может оказаться неразрешимой задачей<sup>1)</sup>.

## 6. СИНХРОННЫЕ ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

С появлением VLSI (СБИС) технологии, в которой один или более процессоров могут быть помещены на четвертьдюймовый чип, естественно представить себе в будущем устройство, составленное из многих тысяч таких процессоров, совместно работающих параллельно для решения одной задачи. Хотя никакой очень большой машины общего назначения (универсальной) такого рода еще не построено, такие замыслы существуют (см. Шварц [72]). Это мотивирует развитие в последнее время очень привлекательной ветви вычислительной сложности: теории крупномасштабных параллельных синхронных вычислений, в которых число процессоров есть ресурс, ограниченный параметром  $H(n)$  ( $H$  — от hardware) таким же образом, что в последовательной теории сложности ограничение памяти параметром  $S(n)$ . Обычно  $H(n)$  — фиксированный полином от  $n$ .

Было предложено немало моделей параллельных вычислений (обзор см. [21]), так же как существует и много конкурирующих последовательных моделей (см. разд. 2). Однако среди них есть две наиболее предпочтительные. Первая — класс моделей:

<sup>1)</sup> Подробнее теорию вероятностных вычислений см. Гилл [32].

с общей памятью, в которой большое число процессоров связано посредством памяти с произвольным доступом, которая объединяет их. Для таких моделей опубликовано много параллельных алгоритмов, так как реальные параллельные машины, когда они будут построены, могут быть очень похожими на них. Однако для математической теории эти модели не вполне удовлетворительны, потому что слишком многие подробности их архитектуры произвольны. Как разрешить конфликты в общей памяти при чтении и записи? Какие базисные операции допустимы для каждого процессора? Следует ли одно обращение к памяти оценивать в  $\log H(n)$  единиц времени?

Поэтому я предпочитаю более чистую модель, обсуждаемую Бородиным [8] (1977), в которой параллельный компьютер есть однородное семейство  $\langle B_n \rangle$  ациклических булевых схем, такое, что  $B_n$  имеет  $n$  входов (и, следовательно, столько же входных наборов длины  $n$ ). Тогда  $H(n)$  (аппаратная сложность) есть просто число элементов в  $B_n$ , а  $T(n)$ , время параллельного вычисления, есть глубина схемы  $B_n$  (т.е. длина длиннейшего пути от входа к выходу). Практическое обоснование этой модели состоит в том, что, вероятно, все реальные машины (включая машины с общей памятью) построены из булевых схем. Кроме того, определение минимальной булевой сложности и глубины, необходимых для вычисления функции, есть естественная математическая задача, и она рассматривалась задолго до возникновения теории параллельных вычислений.

К счастью для этой теории, минимальные значения аппаратной сложности  $H(n)$  и параллельного времени  $T(n)$  не слишком сильно различаются для различных конкурирующих моделей параллельных компьютеров. В частности, имеет место интересный общий факт, верный для всех моделей, доказанный впервые для одной конкретной модели Праттом и Стокмейером [58] (1974) и названный в [33] «тезисом о параллельных вычислениях», а именно, задача может быть решена за полиномиальное время относительно  $T(n)$  параллельной машиной (с неограниченной аппаратной сложностью) тогда и только тогда, когда она может быть решена с полиномиальной памятью относительно  $T(n)$  последовательной машиной (с неограниченным временем).

Один из основных вопросов параллельных вычислений состоит в следующем: какие задачи могут быть решены с использованием многих процессоров существенно быстрее, чем с одним процессором? Николас Пиппенджер [57a] формализовал этот вопрос, определив класс (ныне называемый  $NC$ , «Nick's class») задач, решаемых сверхбыстро (время  $T(n) = O((\log n)^c)$ ) параллельным компьютером с практически прием-

лемой ( $H(n) = n^{o(1)}$ ) аппаратной сложностью. К счастью, класс  $NC$  остается одним и тем же независимо от конкретной выбранной модели параллельного компьютера, и легко видеть, что  $NC$  — подкласс класса  $FP$  функций, вычислимых последовательно за полиномиальное время. Наш неформальный вопрос можно теперь сформулировать так: какие задачи из  $FP$  принадлежат также  $NC$ ?

Мыслимо (хотя и маловероятно), что  $NC = FP$ , так как для доказательства  $NC \neq FP$  потребовалось бы совершить прорыв в теории сложности (см. конец разд. 4.1). Поскольку мы не знаем, как доказать, что функция  $f$ , принадлежащая  $FP$ , не принадлежит  $NC$ , лучше всего попытаться доказать, что  $f$  логарифмически (по памяти) полна для  $FP$ . Это аналог доказательства  $NP$ -полноты задач, и практически это делает почти безнадежными попытки найти сверхбыстрые параллельные алгоритмы для  $f$ . Причина в том, что если  $f$  логарифмически (по памяти) полна для  $FP$  и  $f$  принадлежит  $NC$ , то  $FP = NC$ , что было бы большим сюрпризом.

Немалый прогресс достигнут в классификации задач из  $FP$  по принадлежности их  $NC$  или логарифмической (по памяти) полноте в  $FP$  (конечно, они могут быть не принадлежащими ни одному из этих классов). Первый пример задачи, полной в  $P$ , был предложен в 1973 г. мною в [20], хотя я не формулировал этот результат в терминах полноты. Вскоре после этого Джонс и Лаазер [38] определили это понятие полноты и привели около 5 примеров, включая проблему пустоты для контекстно-свободных грамматик. Возможно, простейшая задача, для которой доказана полнота в  $FP$ , есть так называемая проблема значения схемы [47]: для заданной булевой схемы и значений на ее входах найти значение на выходе. Наиболее интересным для меня является пример, принадлежащий Гольдшлягеру, Шоу и Стейплу [34], — нахождение (четности) максимального потока через данную сеть с (большими) положительными целочисленными пропускными способностями ее ребер. Он интересен тонкостью доказательства полноты. Наконец, я упомянул бы, что линейное программирование полно в  $FP$ . В этом случае трудная часть состоит в доказательстве того, что задача принадлежит  $P$  (см. [43]), после чего полнота устанавливается непосредственно [26].

Среди задач, принадлежность которых классу  $NC$  известна, — четыре арифметические операции ( $+$ ,  $-$ ,  $*$ ,  $\div$ ) над двоичными числами, сортировка, связность графа, матричные операции (умножение, обращение, детерминант, ранг), наибольший общий делитель полиномов, контекстно-свободные языки и нахождение в графе минимального остова (см. [11], [21], [63], [67]). Про размер максимального паросочетания для заданно-



го графа известно, что он принадлежит «случайному»  $NC$  ( $NC$ , в котором допускается бросание монеты), хотя вопрос о том, принадлежит ли хотя бы одному «случайному»  $NC$  задача нахождения фактического максимального паросочетания, пока не решен. Результаты [89] и [76b] предлагают общие методы доказательства принадлежности задач классу  $NC$ .

Наиболее интересной задачей в  $FP$ , про которую неизвестна ни полнота в  $FP$ , ни принадлежность случайному  $NC$ , является нахождение наибольшего общего делителя двух целых чисел. Есть много других интересных задач, которые еще должны быть классифицированы, включая нахождение максимального паросочетания или максимальной клики в графе (см. [88]).

## 7. БУДУЩЕЕ

Позвольте мне снова сказать, что область вычислительной сложности велика, а этот обзор краток. Существуют обширные разделы этой науки, которые я опустил вовсе или едва их коснулся. Приношу свои извинения исследователям в этих областях.

Один относительно новый и волнующий раздел, названный Яо [92] «вычислительной теорией информации», продолжает классическую шенновскую теорию информации, введя в рассмотрение информацию, доступную с помощью практически осуществимого вычисления. Эта тематика стала широко обсуждаться в основном после работ Диффи и Хелмана [25] и Райвеста, Шамира и Адлемана [67a] об общедоступных криптосистемах, хотя ее корни в теории вычислений восходят к Колмогорову [45] и Чайтину [14a], [14b], которые первыми придали смысл понятию «случайности» одной конечной последовательности, используя теорию вычислений. Интересная идея в этой теории, рассмотренная Шамиром [73] и Блюмом и Микали [7], касается порождения псевдослучайных последовательностей, в которых будущие разряды доказуемо трудно предсказать в терминах предшествующих. Яо [92] доказывает, что существование таких последовательностей имело бы положительные последствия в вопросе о детерминированной сложности вероятностного класса  $R$  (см. разд. 5). Фактически вычислительная теория информации обещает пролить свет на роль случайности в вычислении.

В добавление к вычислительной теории информации мы можем ожидать интересные результаты для вероятностных алгоритмов, параллельных вычислений и (при везении) нижних оценок. Что касается нижних оценок, единственный прорыв, на который, по-моему, можно надеяться в недалеком будущем, — это доказательство того, что не всякая задача из  $P$  решается со

сложностью (по памяти)  $O(\log n)$ , и, возможно, также того, что  $P \neq NC$ . Во всяком случае, активность в области вычислительной сложности остается очень высокой, и я с нетерпением жду, что же принесет будущее.

### БЛАГОДАРНОСТИ

Я благодарен своим коллегам по теории сложности в Торонто за многие полезные замечания и предложения, особенно Аллану Воробину, Иоахиму фон цур Гатену, Сильвио Микали и Чарльзу Ракову.

### ЛИТЕРАТУРА

1. Adleman L. Two theorems on random polynomial time. Proc. 19th Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1978), 75—83.
2. Adleman L., Pomerance C., and Rumley R. S. On distinguishing prime numbers from composite numbers. Annals of Math. 117 (January 1983), 173—206.
3. Aho A. V., Hopcroft, J. E., and Ullman, J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974. [Имеется перевод: Ахо А., Хопкрофт Д., Ульман Д. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979.]
4. Bennett J. H. On Spectra. Doctoral dissertation, Department of Mathematics, Princeton University, 1962.
5. Berlecamp E. R. Factoring polynomials over large finite fields. Math. Comp. 24 (1970), 713—735.
6. Blum M. A machine independent theory of the complexity of recursive functions. JACM 14, 2 (April 1967), 322—336. [Имеется перевод: Блюм М. Машинно-независимая теория сложности рекурсивных функций. В сб.: Проблемы математической логики. — М.: Мир, 1970, с. 401—422.]
7. Blum M., and Micali S. How to generate cryptographically strong sequences of pseudo random bits. Proc. 23rd IEEE Symp on Foundations of Computer Science. IEE Computer Society, Los Angeles (1982), 112—117.
8. Borodin A. On relating time and space to size and depth. SIAM J. Comp. 6 (1977), 733—744.
9. Borodin A. Structured vs. general models in computational complexity. In: Logic and Algorithmic. Monographie no. 30 de L'Enseignement Mathematique Universite de Geneve, 1982.
10. Borodin A., and Cook S. A time-space tradeoff for sorting on a general sequential model of computation. SIAM J. Comput. 11 (1982), 287—297.
11. Borodin A., von zur Gathen, J., and Hopcroft, J. Fast parallel matrix and GCD computations. 23rd IEEE Symp. on Foundations of Computer Science. IEEE computer society, Los Angeles (1982), 65—71.
12. Borodin A., and Munro, I. The Computational Complexity of Algebraic and Numeric Problems. Elsevier, New York, 1975.
13. Brockett R. W., and Dobkin, D. On the optimal evaluation of a set of bilinear forms. Linear Algebra and its Applications 19 (1978), 207—235.
- 14a. Chaitin G. J. On the length of programs for computing finite binary sequences. JACM 13, 4 (October 1966), 547—569; JACM 16, 1 (October 1969), 145—159.
- 14b. Chaitin G. J. A theory of program size formally identical to informational theory. JACM 22, 3 (July 1975), 329—340.

15. Cobham A. The intrinsic computational difficulty of functions. Proc. 1964 International Congress for Logic, Methodology, and Philosophy of Sciences. Y. Bar-Hellel, Ed., North Holland, Amsterdam, 1965, 24—30.
16. Cobham A. The recognition problem for the set of perfect squares. IEEE Conference Record Seventh SWAT (1966), 78—87.
17. Cohen H., and Lenstra, H. W., Jr. Primarily testing and Jacobi sums. Report 82—18, University of Amsterdam, Dept. of Math., 1982.
18. Cook S. A. The complexity of Theorem proving procedures. Proc. 3rd ACM Symp. on Theory of Computing. Shaker Heights, Ohio (May 3—5, 1971), 151—158. [Имеется перевод: Кук С. А. Сложность процедур вывода теорем. Кибер. сб., н. с., вып. 12. — М.: Мир, 1975, с. 5—16.]
19. Cook S. A. Linear time simulation of deterministic two-way pushdown automata. Proc. IFIP Congress 71 (Theoretical Foundations). North Holland, Amsterdam, 1972, 75—80.
20. Cook S. A. An observation on time-storage tradeoff. JCSS 9 (1974), 308—316. Первоначально опубликовано в Proc. 5th ACM Symp. on Theory of Computing. Austin, TX (April 30 — May 2, 1973), 29—33.
21. Cook S. A. Toward a complexity theory of synchronous parallel computation. L'Enseignement Mathematique XXVII (1981), 99—124.
22. Cook S. A., and Aanderaa, S. O. On the minimum computation time of functions. Trans. AMS 142 (1969), 291—314. [Имеется перевод: Кук С. А., Аандереа С. О. О минимальном времени вычисления функций. Кибер. сб., н. с., вып. 8. — М.: Мир, 1971, с. 168—200.]
23. Cooley J. M., and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. Math. Comput. 19 (1965), 297—301.
24. Coppersmith D., and Winograd, S. On the asymptotic complexity of matrix multiplication. SIAM J. Comp. 11 (1982), 472—492.
25. Diffie W., and Hellman, M. E. New direction in cryptography. IEEE Trans. on Inform. Theory IT-22, 6 (1976), 644—654.
26. Dobkin D., Lipton, R. J., and Reiss, S. Linear programming is log-space hard for P. Inf. Processing Letters 8 (1979), 96—97.
27. Edmonds J. Paths, trees, flowers. Canad. J. Math. 17 (1965), 49—67.
28. Edmonds J. Minimum partition of a matroid into independent subsets. J. Res. Nat. Bur. Standards Sect. B, 69 (1965), 67—72.
29. Ferrante J., and Rackoff, C. W. The Computational Complexity of Logical Theories. Lecture Notes in Mathematics. # 718, Springer Verlag, New York, 1979.
30. Fisher M. J., and Rabin, V. O. Super-exponential complexity of Pressburger arithmetic. In Complexity of Computation. SIAM-AMS Proc. 7, R. Karp, Ed., 1974, 27—42.
31. Garey M. R., and Jonson, D. S. Computers and Intractability: A Guide to the Theory of NP-completeness. W. H. Freeman, San Francisco, 1979. [Имеется перевод: Гэри Майкл Р., Джонсон Дэвид С. Вычислительные машины и труднорешаемые задачи. — М.: Мир, 1982, 416 с.]
32. Gill J. Computational complexity of probabilistic Turing machines. SIAM J. Comput. 6 (1977), 675—695.
33. Goldschlager L. M. Synchronous Parallel Computation. Doctoral dissertation, Dept. of Computer Science, Univ. of Toronto, 1977. См. также JACM, 2, 4 (October 1982), 1073—1086.
34. Goldschlager L. M., Shaw, R. A., and Staples, J. The maximum flow problem is log space complete for P. Theoretical Computer Science 21 (1982), 105—111.
35. Grzegorzczak A. Some classes of recursive functions. Rozprawy Matematyczne, 1963. [Имеется перевод: Гжегорчик А. Некоторые классы рекурсивных функций. В сб.: Проблемы математической логики. — М.: Мир, 1970, с. 9—49.]
36. Hartmanis J. Observations about the development of theoretical computer science. Annals Hist. Comput. 3, 1 (Jan. 1981), 42—51.

37. Hartmanis J., and Stearns, R. E. On the computational complexity of algorithms. Trans. AMS 117 (1965), 285—306. [Имеется перевод: Хартманис Дж., Стирнс Р. О вычислительной сложности алгоритмов. Кибер. сб., н.с., вып. 4. — М.: Мир, 1967, с. 57—85.]
38. Jones N. D., and Laazer, W. T. Complete problems for deterministic polynomial time. Theoretical Computer Science 3 (1977), 105—127.
39. Kaltofen E. A polynomial reduction from multivariate to bivariate integer polynomial factorization. Proc. 14th ACM Symp. in Theory Comp., San Francisco, CA (May 5—7, 1982), 261—266.
40. Kaltofen E. A polynomial-time reduction from bivariate to univariate integral polynomial factorization. Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1982), 57—64.
41. Карацуба А., Офман Ю. Умножение многозначных чисел на автоматах. — ДАН СССР 145, 2 (1962), 293—294.
42. Karp R. M. Reducibility among combinatorial problems. In: Complexity of Computer computations. R. E. Miller and J. W. Thatcher, Eds., New York, 1972, 85—104. [Имеется перевод: Карп Р. М. Взаимная сводимость комбинаторных проблем. Кибер. сб., н.с., вып. 12. — М.: Мир, 1975, с. 16—38.]
43. Хачиян Л. Г. Полиномиальный алгоритм для линейного программирования. ДАН СССР 224, 5 (1979), с. 1093—1096.
44. Knuth D. E. The Art of Computer Programming, Vol. 3 Sorting and Searching. Addison-Wesley, Reading, MA, 1973. [Имеется перевод: Кнут Д. Е. Искусство программирования для ЭВМ, т. 3. Сортировка и поиск. — М.: Мир, 1978.]
45. Колмогоров А. Н. Три подхода к определению понятия «количество информации». Проблемы передачи информации 1, 1 (1965), с. 3—11.
46. Колмогоров А. Н., Успенский В. А. К определению понятия алгоритма. УМН 13 (1958), с. 3—28.
47. Ladner R. E. The circuit value problem is log space complete for P. SIGAST News 7, 1 (1975), 18—20.
48. Lenstra A. K., Lenstra, H. W., and Lovasz, L. Factoring polynomials with rational coefficients. Report 82—05, University of Amsterdam, Dept. of Math., 1982.
49. Левин Л. А. Универсальные задачи перебора. Проблемы передачи информации 9 (1973), с. 115—116.
50. Luks E. M. Isomorphism of graphs of bounded valence can be tested in polynomial time. Proc. 21st Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1980), 42—49.
51. Meyer A. R. Weak monadic second-order theory of successor is not elementary-recursive. Lecture Notes in Mathematics 453. Springer Verlag, New York, 1975. 132—154. [Имеется перевод: Мейер А. Р. Слабая сингулярная теория второго порядка функционирования не элементарно разрешима. Кибер. сб., н.с., вып. 12. — М.: Мир, 1975, с. 62—77.]
52. Meyer A. R., and Stockmeyer L. J. The equivalence problem for regular expressions with squaring requires exponential space. Proc. 13th IEEE Symp. on Switching and Automata Theory (1972), 125—129.
53. Miller G. L. Riemann's hypothesis and tests for primality. J. Comp. System Sci. 13(1976), 300—317.
54. Oppen D. C. A  $2^{2^{pn}}$  upper bound on the complexity of Presburger arithmetic. J. Comp. Syst. Sci. 16 (1978), 323—332.
55. Papadimitriou C. H., and Steiglitz, K. Combinatorial Optimization Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ, 1982. [Имеется перевод: Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. — М.: Мир, 1985.]
56. Paterson M. S., Fischer M. J., and Meyer A. R. An improved overlap argument for on-line multiplication. SIAM—AMS Proc. 7, Amer. Math. Soc., Providence, 1974, 97—111.

- 57a. Pippenger N. On simultaneous resource bounds (preliminary version). Proc. 20th IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1979), 307—311.
- 57b. Pippenger N. J., and Fischer, M. J. Relations among complexity measures. JACM 26, 2 (April 1979), 361—381.
58. Pratt V. A., and Stockmeyer L. J. A characterization of the power of vector machines. J. Comput. System Sci. 12 (1976), 198—221. Originally in Proc. 6th ACM Symp. on Theory of Computing, Seattle, WA (April 30—May 2, 1974), 122—134.
59. Rabin M. O. Speed of computation and classification of recursive sets. Third Convention Sci. Soc., Israel, 1959, 1—2.
60. Rabin M. O. Degree of difficulty of computing a function and a partial ordering of recursive sets. Tech. Rep. No. 1, O. N. R, Jerusalem, 1960.
61. Rabin M. O. Probabilistic algorithms. In Algorithms and Complexity, New Directions and Recent Trends, J. F. Traub, Ed. Academic Press, New York, 1976, 29—39.
62. Rabin M. O. Complexity of computations. Comm. ACM 20, 9 (September 1977), 625—633.
63. Reif J. H. Symmetric complementation. Proc. 14th ACM Symp. on Theory of Computing, San Francisco, CA (May 5—7, 1982), 201—214.
64. Reisch S., and Schnitger, G. Three applications of Kolmogorov complexity. Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1982), 45—52.
65. Ritchie R. W. Classes of Recursive Functions of Predictable Complexity. Doctoral Dissertation, Princeton University, 1960.
66. Ritchie R. W. Classes of predictably computable functions. Trans AMS 106 (1963), 139—173. [Имеется перевод: Ричи Р. В. Классы предсказуемо вычислимых функций. — Проблемы математической логики. — М.: Мир, 1970, с. 50—93.]
- 67a. Rivest R. L., Shamir A., and Adleman L. A method for obtaining digital signatures and public-key cryptosystems. Comm. ACM 21, 2 (February 1978), 120—126.
- 67b. Ruzzo W. L. On uniform circuit complexity. J. Comput. System Sci. 22 (1981), 365—383.
- 68a. Savage J. E. The Complexity of Computing. Wiley, New York, 1976.
- 68b. Schnorr C. P. The network complexity and the Turing machine complexity of finite functions. Acta Informatica 7 (1976), 95—107.
69. Schönhage A. Storage modification machines. SIAM J. Comp. 9 (1980), 490—508.
70. Schönhage A., and Strassen V. Schnelle Multiplikation grosser Zahlen. Computing 7 (1971), 281—292. [Имеется перевод: Шёнхаге А., Штрассен В. Быстрое умножение больших чисел. Кибер. сб., н. с., вып. 10. — М.: Мир, 1973, с. 87—98.]
71. Schwartz J. T. Probabilistic algorithms for verification of polynomial identities. JACM 27, 4 (October 1980), 701—717.
72. Schwartz J. T. Ultracomputers. ACM Trans. on Prog. Languages and Systems 2, 4 (October 1980), 484—521.
73. Shamir A. On the generation of cryptographically strong pseudo random sequences. 8th Int. Colloquium on Automata, Languages, and Programming (July 1981). Lecture Notes in Computer Science 115. Springer Verlag, New York, 544—550.
74. Shannon C. E. The synthesis of two terminal switching circuits. BSTJ 28 (1949), 59—98. [Имеется перевод: Шеннон К. Синтез двухполюсных переключательных схем. В кн.: Шеннон К. Работы по теории информации и кибернетике. — М.: ИЛ, 1962, с. 59—105.]
75. Smale S. On the average speed of the simplex method of linear programming. Preprint, 1982.

76. Smale S. The problem of the average speed of the simplex method. Preprint, 1982.
77. Solovay R., and Strassen V. A fast monte-carlo test for primality. *SIAM J. Comput.* 6 (1977), 84—85.
78. Stearns R. E., Hartmanis J., and Lewis, P. M. II Hierarhies of memory limited computations. 6th IEEE Symp. on Switching Circuit Theory and Logical Design (1965), 179—190. [Имеется перевод: Стирнз Р. Е., Хартманис Дж., Льюис П. М. II. Иерархии вычислений с ограниченной памятью. В сб.: Проблемы математической логики. — М.: Мир, 1970, с. 301—319.]
79. Stockmeyer L. J. The complexity of decision problems in automata theory and logic. Doctoral Thesis, Dept. of Electrical Eng., MIT, Cambridge, MA., 1974; Report TR-133, MIT, Laboratory for Computing Science.
80. Stockmeyer L. J. Classifying the computational complexity of problems. Research Report RC 7606 (1979), Math Sciences Dept., IBM T. J. Watson Research Center, Yorktown Heights, N. Y.
81. Strassen V. Gaussian elimination is not optimal. *Num. Math.* 13 (1969), 354—356. [Имеется перевод: Штрассен В. Алгоритм Гаусса не оптимален. Кибер. сб., н. с., вып. 7. — М.: Мир, 1970, с. 67—70.]
82. Strassen V. Die Berechnungskomplexitat von elementarsymmetrischen Funktionen und Interpolationskoeffizienten. *Numer. Math.* 20 (1973), 238—251. [Имеется перевод: Штрассен Ф. Сложность вычисления элементарных симметрических функций и коэффициентов интерполяционного полинома. Кибер. сб., н. с., вып. 15. — М.: Мир, 1978, с. 5—21.]
83. Baur W., and Strassen V. The complexity of partial derivatives. Preprint, 1982.
84. Тоом А. Л. О сложности схемы из функциональных элементов, реализующей умножение целых чисел. *ДАН СССР* 150, 3 (1963), с. 496—498.
85. Turing A. M. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc. ser. 2*, 42 (1937), 230—265. A correction, *ibid.* 43 (1937), 544—546.
86. Valiant L. G. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8 (1979), 189—202.
87. Valiant L. G. The complexity of computing the permanent. *Theoretical Computer Science* 8 (1979), 189—202. [Имеется перевод: Вэлиант Дж. Сложность вычисления перманента. Кибер. сб., н. с., вып. 18. — М.: Мир, 1981, с. 125—140.]
88. Valiant L. G. Parallel computation. *Proc. 7th IBM Japan Symp. Academic 6 Scientific Programs, IBM Japan, Tokyo* (1982).
89. Valiant L. G., Skyum S., Berkowitz S., and Rackoff, C. Fast parallel computation on polynomials using few processors. Preprint. (Preliminary version in *Springer Lecture Notes in Computer Science* 118 (1981), 132—139).
90. von Neumann J. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games II*. H. W. Kahn and A. W. Tucker, Eds. Princeton Univ. Press, Princeton, NJ, 1953.
91. Yamada H. Real time computation and recursive functions not real-time computable. *IRE Transactions on Electronic Computers* EC-11 (1962), 753—760. [Имеется перевод: Ямада Х. Вычисления в реальное время и рекурсивные функции, не вычислимые в реальное время. В кн.: Проблемы математической логики. — М.: Мир, 1970, с. 139—155.]
92. Yao, A. C. Theory and applications of trapdoor functions (extended abstract). *Proc. 23rd IEEE Symp. on Foundations of Computer Science*. IEEE Computer Society, Los Angeles (1982), 80—91.

1985

## Комбинаторика, сложность и случайность

*Ричард М. Карп*

Ричард М. Карп из Калифорнийского университета, Беркли, получил в 1985 г. премию АСМ им. А. М. Тьюринга за свои фундаментальные достижения в теории сложности. Эта премия, являющаяся высшей наградой АСМ за исследования по информатике, была вручена на ежегодной конференции Ассоциации в Денвере, в октябре.

К 1972 г. Карп приобрел репутацию одного из ведущих теоретиков информатики в основном из-за своей новаторской статьи «Сводимость комбинаторных задач» (в книге *Complexity of Computer Computations (Symposium Proceedings)*, Plenum, New York, 1972). Продолжая более раннюю работу Стивена Кука, он применил понятие сводимости в полиномиальное время и показал, что если только  $P$  не совпадает с  $NP$ , большинство классических задач комбинаторной оптимизации  $NP$ -полны и, следовательно, практически неразрешимы. Это изменило подход специалистов по информатике к практическим задачам, таким, как выбор маршрута (включая знаменитую задачу о коммивояжере), упаковка, покрытие, паросочетания, разбиение и составление расписаний, и усилило внимание к приближенным методам решения этих трудных задач. Потом в своей работе Карп впервые применил вероятностный анализ, чтобы обосновать эффективность таких приближенных методов.

Р. Карп — профессор трех факультетов в Бэркли: электротехники и информатики, математики, а также инженерного дела и исследования операций. В этом году он сопредседатель одногодичной исследовательской программы по вычислительной сложности в Математическом институте (Mathematical Sciences Research Institute), финансируемой Национальным научным фондом.

Родился в Бостоне, степень доктора философии по прикладной математике получил в Гарвардском университете в 1959 г. Он занимался девять лет информатикой в исследовательской лаборатории IBM в Йорктаун Хайтс, Нью-Йорк, и преподавал в Мичиганском, Колумбийском и Нью-Йоркском университетах и Политехническом институте в Бруклине. С 1968 г. он преподает в Беркли и получил профессорство Миллера в Беркли.

в 1980—1981 гг. В 1980 г. был избран в Национальную академию наук.

Лауреат Тьюринговской премии за 1985 г. представил свой обзор развития исследований в области, которая сейчас стала называться теоретической информатикой.

\* \* \*

Эта лекция посвящена памяти моего отца Абрахама Луиса Карпа.

Для меня честь и удовольствие быть лауреатом Тьюринговской премии этого года. Какое бы удовлетворение ни доставило мне такое признание, я нахожу, что мое величайшее удовлетворение как исследователя связано с выполнением самого исследования и с дружескими связями, которые я при этом приобрел. Я хотел бы описать вам все 25 лет исследований в области комбинаторных алгоритмов и вычислительной сложности и рассказать о некоторых понятиях, которые мне кажутся важными, и о некоторых людях, которые меня вдохновляли и влияли на меня.

## НАЧАЛО

Я пришел в информатику довольно случайно. По окончании в 1955 г. Гарвардского колледжа с дипломом по математике я стал думать, что же делать дальше. Зарабатывать на жизнь было малопривлекательным, и очевидный выбор пал на аспирантуру. Одна из возможностей была продолжать занятия математикой, но тогда все увлекались абстрактностью и общностью, а конкретная и прикладная математика, вдохновлявшая меня, казалось, была не в моде.

Итак, почти не имея другого выбора, я приступил к программе подготовки диссертации доктора философии в Гарвардской вычислительной лаборатории. Большинство предметов, которые теперь стали непременными составляющими курсов информатики, тогда и в мыслях ни у кого не было, и я прослушал эклектичный набор дисциплин: теория переключательных схем, численный анализ, прикладная математика, вероятность и статистика, исследование операций, электроника и математическая лингвистика. Хотя программа оставляла желать лучшего в смысле глубины и согласованности, там была особая атмосфера: мы знали, что присутствием при рождении новой науки, изучающей компьютеры и их применения. Я обнаружил, что нахожу красоту и элегантность в структуре алгоритмов и люблю дискретную математику, составлявшую основу для изучения компьютеров и вычислений. Короче, я попал более или менее случайно в научную область, которая мне очень понравилась.



## ЛЕГКИЕ И ТРУДНЫЕ КОМБИНАТОРНЫЕ ЗАДАЧИ

Уже с той ранней поры я приобрел особый интерес к комбинаторным задачам поиска, — задачам, которые можно уподобить головоломкам, где нужно собрать определенным образом отдельные части в одну общую структуру. Такие задачи включают поиск в конечном, но очень большом структурированном множестве возможных решений, образов или конфигураций с целью найти одно решение, удовлетворяющее заданному множеству условий. Вот несколько примеров таких задач: размещение и соединение компонент интегральных схем, составление расписания игр национальной футбольной лиги и управление движением школьных автобусов.

Во всякой из этих комбинаторных головоломок таится возможность комбинаторного взрыва. По причине огромного, неудержимо растущего числа возможностей, возникающих при поиске, можно столкнуться с большим объемом вычислений, если не применить какую-нибудь тонкость при поиске в пространстве возможных решений. Я хотел бы начать техническую часть своей беседы с рассказа о первых моих встречах с комбинаторными взрывами.

Первое поражение при столкновении с этим явлением я потерпел вскоре после того, как поступил в Исследовательский центр IBM на Йорктаун Хайтс в 1959 г. Я был назначен в группу, возглавляемую Дж. Ротом, известным алгебраическим топологом, который внес заметный вклад в теорию переключательных схем. Задача нашей группы состояла в создании компьютерной программы для автоматического синтеза переключательных схем. На входе программы было множество булевых формул, описывающих, как выходы схемы зависят от ее входов;

программа должна была строить схему, состоящую из минимального числа логических элементов. На рис. 1 показана схема для функции голосования от трех переменных, выход принимает значение единица, когда по крайней мере 2 из 3 переменных  $x$ ,  $y$  и  $z$  принимают значение единица.

Программа, которую мы разработали, содержала много элегантных упрощений и ухищрений, но ее основной механизм состоял в:

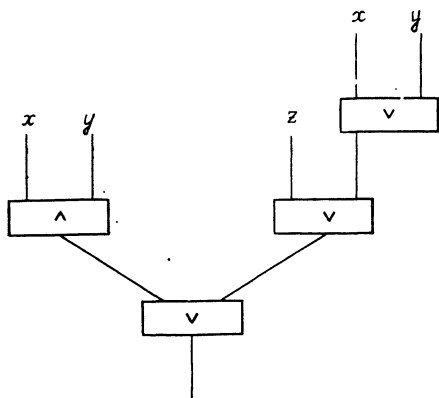


Рис. 1. Схема для функции голосования.

перечислении возможных схем в порядке возрастания их стоимости (сложности). Число схем, которые эта программа должна была просматривать, страшно росло по мере роста числа входных переменных, и, как следствие, мы не смогли продвинуться далее решения игрушечных задач. Сегодня наш оптимизм в самой попытке применить перечислительный подход может показаться крайне наивным, но мы были не единственными, кто попал в эту ловушку, многие работы по автоматическому доказательству теорем за последние два десятилетия начинались с прилива энтузиазма от успешного решения игрушечных задач, за которым следовало разочарование, как только становилась очевидной вся серьезность явления комбинаторного взрыва.

Примерно в то же самое время я начал работать над задачей о коммивояжере с Майклом Хелдом из ИВМ. Эта задача получила свое название от ситуации коммивояжера, который хочет посетить все города на своей территории, начиная и кончая путь в родном городе и минимизируя при этом общую стоимость путешествия. В специальном случае, когда города — точки на плоскости и стоимость путешествия равна евклидову расстоянию, задача состоит просто в нахождении многоугольника с минимальным периметром, проходящим через все города (рис. 2). Несколькими годами ранее Джордж Данциг, Раймонд Фалкерсон и Селмер Джонсон из Рэнд корпорейшн, используя комбинацию ручных и автоматических вычислений, преуспели в решении задачи для 49 городов, и мы надеялись побить этот рекорд.

Несмотря на невинный вид, задача о коммивояжере потен-

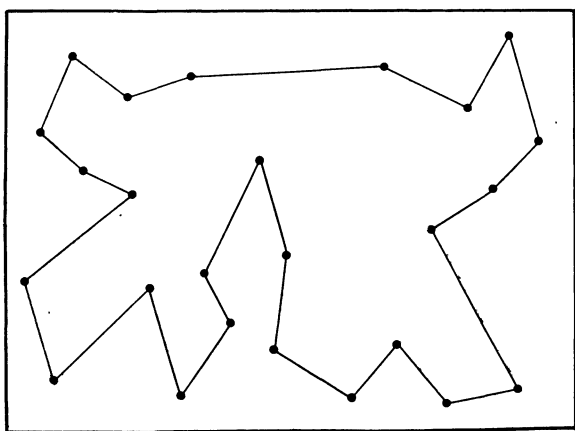


Рис. 2. Маршрут коммивояжера.

циально способна привести к комбинаторному взрыву, так как число возможных маршрутов на плоскости через  $n$  городов равно  $(n-1)!/2$ , очень быстро растущей функции  $n$ . Например, когда число городов равно всего лишь 20, время, требуемое для бесхитростного перечисления всех возможных маршрутов со скоростью миллион маршрутов в секунду, составит больше тысячи лет.

Хелд и я испробовали ряд подходов к задаче о коммивояжере. Мы начали с переоткрытия сокращения, основанного на динамическом программировании, на которое указал Ричард Беллман. Метод динамического программирования уменьшает время поиска до  $n^2 2^n$ , но эта функция также быстро растет, и метод ограничен практически задачами не более чем с 16 городами. На некоторое время мы оставили идею решить задачу точно и экспериментировали с методами локального поиска, которые приводят к достаточно хорошим, но не оптимальным маршрутам. При этих методах начинают с произвольного маршрута, снова и снова отыскивая локальные изменения, улучшающие его. Процесс продолжается, покуда не найден маршрут, который локально не может быть улучшен. Наши методы локальных изменений были довольно неуклюжими, Чень Линь и Брайан Керниган из лаборатории Бэлл позднее нашли гораздо лучшие. Такие быстрые и грубые методы часто весьма полезны на практике, если не требуется найти строго оптимальное решение, но никогда нельзя гарантировать, насколько хорошо они будут работать.

Затем мы начали исследовать методы ветвей и границ. Такие методы по сути перечислительные, но они выигрывают в эффективности в результате отсеечения больших областей пространства возможных решений. Это делается путем вычисления нижней оценки стоимости каждого маршрута, включающего некоторые связи и исключающего некоторые другие; если нижняя оценка достаточно велика, то отсюда следует, что такой маршрут не может быть оптимальным. После длинной серии безуспешных экспериментов мы с Хелдом случайно обнаружили сильный метод получения нижних оценок. Эта техника ограничений позволила нам сильно сокращать пространство поиска, так что мы сумели решать задачи даже с 65 городами. Я не думаю, что какой-нибудь из моих теоретических результатов потряс меня столь же сильно, как вид чисел, появляющихся из компьютера ночью, когда Хелд и я впервые испытывали наш метод ветвей и границ. Позднее мы обнаружили, что наш метод — не что иное, как вариант старой техники, именуемой релаксациями Лагранжа, которая теперь рутинно используется для построения нижних оценок в методе ветвей и границ.

Короткое время наша программа была мировым чемпионом

по решению задачи о коммивояжере, но в наши дни существуют гораздо более впечатляющие программы. Они базируются на технике, называемой полиэдральной комбинаторикой, которая пытается свести частные случаи задачи о коммивояжере к очень большим задачам линейного программирования. Такие методы могут решать задачи более чем с 300 городами, но такой подход не полностью исключает комбинаторные взрывы, так как время, требуемое для решения задачи, продолжает расти экспоненциально как функция числа городов.

Задача о коммивояжере остается очаровательной загадкой. Недавно опубликована книга объемом более чем 400 страниц, покрывающая большинство из того, что известно об этой непростой задаче. Позднее мы обсудим теорию *NP*-полноты, которая дает необходимое обоснование того, что задача о коммивояжере по самой своей природе практически нерешаема, так что независимо от конструкции алгоритмов нельзя полностью исключить возможности комбинаторных взрывов, вспыхивающих внутри этой задачи.

В начале 1960-х годов лаборатория IBM на Йорктаун Хайтс имела превосходную группу специалистов по комбинаторике, и под их руководством я научился важным методам решения некоторых комбинаторных задач такого рода, позволяющим избежать комбинаторных взрывов. Например, я познакомился со знаменитым простым алгоритмом Данцига для линейного программирования. Задача линейного программирования состоит в нахождении точки на многограннике в многомерном пространстве, ближайшей к данной внешней гиперплоскости (многогранник — это обобщение многоугольника в двумерном пространстве или обычного многогранного тела в трехмерном пространстве, а гиперплоскость — это обобщение прямой на плоскости или плоскости в трехмерном пространстве). Ближайшая к гиперплоскости точка — всегда угловая точка, или вершина многогранника (рис. 3). На практике симплекс-метод очень быстро позволяет найти нужную вершину.

Я также изучал прекрасную теорию потоков в сетях Лестера Форда и Фалкерсона. Эта теория описывает, с какой скоростью некую субстанцию, например нефть, газ, электри-

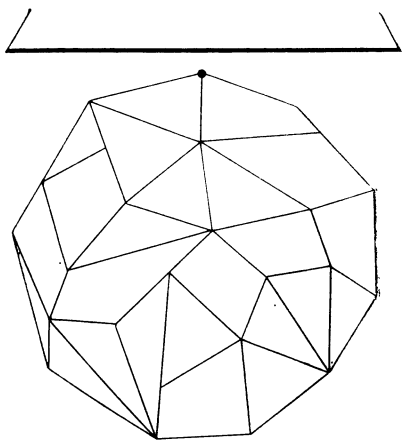


Рис. 3. Задача линейного программирования.

чество или биты информации, можно прогонять через сеть, в которой каждое ребро имеет ограниченную пропускную способность. Многие комбинаторные задачи, которые на первый взгляд кажутся не имеющими никакого отношения к потокам, протекающим через сети, могут быть переформулированы как задачи в потоках в сетях, и эта теория способствует элегантному и эффективному решению таких задач без использования других арифметических операций, кроме сложения и вычитания.

Позвольте мне проиллюстрировать эту красивую теорию наброском так называемого венгерского алгоритма для решения задачи комбинаторной оптимизации, известной как задача марьяжа. Эта задача относится к обществу из  $n$  мужчин и  $n$  женщин. Задача состоит в разбиении на пары мужчин и женщин способом с наименьшими затратами, где стоимость каждого сочетания задана. Эти стоимости заданы матрицей  $n \times n$ , в которой каждая строка соответствует одному из мужчин и каждый столбец — одной из женщин. Вообще каждое паросочетание  $n$  мужчин с  $n$  женщинами соответствует выбору  $n$  элементов из матрицы, никакие два из которых не принадлежат одному столбцу и одной строке, стоимость паросочетания есть сумма  $n$  выбранных элементов. Число возможных паросочетаний есть  $n!$ ; эта функция растет настолько быстро, что простой перебор будет малоэффективен. На рис. 4 (а) приведен пример  $3 \times 3$ -матрицы, в котором мы видим, что стоимость паросочетания мужчины номер 3 с женщиной номер 2 равна 9, элементу третьей строки и второго столбца данной матрицы. Ключевое наблюдение, лежащее в основе венгерского алгоритма, состоит в том, что задача не изменяется, если из всех элементов одной строки матрицы вычесть одну и ту же константу. Используя эту свободу модификации матрицы, алгоритм пытается построить матрицу, в которой все элементы неотрицательны, так что каждое полное паросочетание имеет неотрицательную общую стоимость, и в которой существует полное паросочетание со всеми нулевыми элементами. Такое паросочетание, очевидно, оптимально для построенной матрицы стоимости, и оно оптимально также для исходной матрицы. В нашем примере  $3 \times 3$ -матрицы алгоритм начинает с вычитания наименьшего элемента каждой строки из всех остальных элементов этой строки, после чего образуется матрица, в которой каждая строка со-

$$\begin{array}{cccc}
 \begin{bmatrix} 3 & 4 & 2 \\ 8 & 9 & 1 \\ 7 & 9 & 5 \end{bmatrix} & 
 \begin{bmatrix} 1 & 2 & 0 \\ 7 & 8 & 0 \\ 2 & 4 & 0 \end{bmatrix} & 
 \begin{bmatrix} 0 & 0 & 0 \\ 6 & 6 & 0 \\ 1 & 2 & 0 \end{bmatrix} & 
 \begin{bmatrix} 0 & \textcircled{0} & 1 \\ 5 & 5 & \textcircled{0} \\ \textcircled{0} & 1 & 0 \end{bmatrix} \\
 \text{(a)} & \text{(b)} & \text{(c)} & \text{(d)}
 \end{array}$$

Рис. 4. Пример задачи паросочетания.

держит по крайней мере один нуль (рис. 4 (b)). Затем, чтобы получить нуль в каждом столбце, алгоритм вычитает из всех элементов каждого столбца, который еще нуля не содержит, наименьший элемент этого столбца (рис. 4 (c)). В этом примере все нули полученной матрицы лежат в первой строке или третьем столбце, так как полное паросочетание содержит точно один элемент из каждого столбца и каждой строки; этого недостаточно для нахождения полного паросочетания, состоящего целиком из нулевых элементов. Чтобы построить такое паросочетание, необходимо получить нуль в нижней левой части матрицы. В данном случае алгоритм строит такой нуль вычитанием 1 из первого и второго столбцов и добавлением единицы к первой строке (рис. 4 (d)). В полученной неотрицательной матрице три обведенных кружками элемента дают полное паросочетание нулевой стоимости, и это паросочетание, следовательно, оптимально как в итоговой, так и в исходной матрицах.

Этот алгоритм гораздо тоньше и эффективнее, чем простой перебор. Время, затрачиваемое им на решение проблемы марьяжа, растет только как третья степень (куб)  $n$ , числа строк и столбцов матрицы, и как следствие, можно решать примеры с тысячами строк и столбцов.

Поколение исследователей, основавших теорию линейного программирования и теорию потоков в сетях, заняло прагматическую позицию по отношению к задачам сложности вычислений: алгоритм рассматривался как эффективный, если он быстро работал на практике, и не так уж важно было доказывать, что он быстр во всех возможных случаях. В 1967 г. я заметил, что стандартный алгоритм для решения некоторых задач о потоках в сетях имеет теоретический изъян, из-за чего он очень медленно работает на некоторых специально подобранных примерах. Я нашел, что нетрудно исправить этот изъян, и рассказал об этом результате на семинаре по комбинаторике в Принстоне. Принстонцы сообщили мне, что Джек Эдмондс из Национального бюро стандартов представлял очень похожие результаты на том же семинаре на предыдущей неделе.

В результате такого совпадения мы с Эдмондсом начали работать вместе над теоретической эффективностью алгоритмов для потоков в сетях и затем написали совместную статью. Но главный результат нашего сотрудничества состоял в укреплении моей приверженности некоторым идеям о вычислительной сложности, к которым я уже ощупью двигался и которым предстояло оказать сильное влияние на дальнейшее направление моих исследований. Эдмондс был искусным практиком и умело использовал идеи, относящиеся к линейному программированию, при разработке прекрасных алгоритмов для ряда комбинаторных задач. Но вдобавок к его искусности в построении алго-

ритмов он определил своих современников в другом важном отношении. Он разработал ясное и точное понимание того, что означает для произвольного алгоритма «быть эффективным». Его статьи сделали общепринятой ту точку зрения, что алгоритм нужно рассматривать как «хороший», если время его выполнения ограничено полиномиальной функцией от размера входного слова (числа входов), а не, скажем, экспоненциальной функцией. Например, согласно эдмондсовской концепции, венгерский алгоритм для задачи марьяжа — хороший алгоритм, потому что время его работы растет как куб размера входа. Но, насколько мы знаем, может не существовать хорошего алгоритма для задачи о коммивояжере, потому что все алгоритмы, которые мы испытывали, имели экспоненциальную зависимость времени выполнения от размера задачи. Определение Эдмондса дало ясный критерий разграничения легких и трудных комбинаторных задач и впервые открыло, по крайней мере в моем понимании, возможность того, что мы однажды можем прийти к теореме, которая докажет или опровергнет предположение, что задача о коммивояжере по своей природе практически неразрешима.

## ПУТЬ К NP-ПОЛНОТЕ

Параллельно с развитием событий в области комбинаторных алгоритмов набирал силу в течение 60-х годов другой важный поток исследований — теория вычислительной сложности. Основы этого предмета были заложены в 30-е годы группой логиков, включающей Аллана Тьюринга, которых заинтересовало существование или несуществование автоматических процедур для установления того, истинны или ложны математические утверждения.

Тьюринг и другие первооткрыватели теории вычислимости были первыми, кто доказал, что некоторые корректно определенные математические задачи неразрешимы, т.е. что в принципе не существует алгоритма, способного к решению всех частных случаев таких задач. Первым примером такой задачи была задача остановки, по существу — вопрос об отладке компьютерных программ. Вход задачи остановки есть компьютерная программа вместе с ее входными данными; задача состоит в том, чтобы решить, будет ли исполнение программы в конце концов завершено. Как может не быть алгоритма для такой корректно определенной задачи? Сложности возникают из-за возможности неограниченного поиска. Очевидное решение состоит в выполнении программы до остановки. Но в какой момент становится логичным прервать исполнение программы, решив, что она никогда не остановится сама? Похоже, что

нет никакого способа установить предел объема необходимых поисков. Используя так называемый диагональный метод, Тьюринг построил доказательство того, что не существует алгоритма, который может успешно рассмотреть все частные случаи проблемы остановки.

Годы шли, и неразрешимые задачи были найдены почти во всех областях математики. Вот пример из теории чисел — задача решения диофантовых уравнений: для данного полиномиального уравнения, скажем

$$4xy^2 + 2xy^2z^3 - 11x^3y^2z^2 = -1164,$$

определить, существует ли его решение в целых числах. Задача нахождения общей разрешающей процедуры для решения таких диофантовых уравнений была впервые поставлена Давидом Гильбертом в 1900 г. и стала известна как 10-я проблема Гильберта. Проблема оставалась открытой до 1971 г., когда было доказано, что такой разрешающей процедуры существовать не может.

Один из фундаментальных инструментов, используемых в проведении границы между разрешимыми и неразрешимыми задачами, — понятие сводимости, которое было впервые выдвинуто на первый план благодаря работе логика Эмиля Поста. Задача *A* сводима к задаче *B*, если, исходя из процедуры, способной решить задачу *B*, можно построить алгоритм для решения задачи *A*. Например, большое значение имел следующий результат: проблема остановки сводима к 10-й проблеме Гильберта. Отсюда следует, что 10-я проблема Гильберта должна быть неразрешимой, так как в противном случае мы могли бы использовать это сведение, чтобы построить алгоритм для проблемы остановки, которая, как известно, неразрешима. Понятие сводимости вновь возникнет, когда мы будем обсуждать Р-полноту и проблему  $P=NP$ .

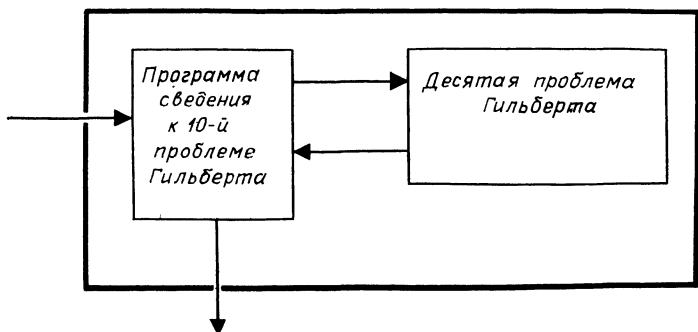


Рис. 5. Проблема остановки сводима к 10-й проблеме Гильберта.



Другой важной темой, которую теория сложности унаследовала от теории вычислимости, является различие между способностью решить задачу и способностью проверить решение. Даже несмотря на то, что не существует общего метода нахождения решения для диофантова уравнения, легко проверить предлагаемое решение. Например, чтобы проверить, является ли  $x=3$ ,  $y=2$ ,  $z=-1$  решением приведенного выше диофантова уравнения, просто подставляют в него эти значения и выполняют арифметические действия. Как мы увидим ниже, в различии между решением и проверкой заключается проблема  $P=NP$ .

Некоторые из самых старых областей теоретической информатики возникли из абстрактных машин и других формализмов теории вычислений. Одна из наиболее важных среди этих областей — теория сложности вычислений. Вместо того чтобы просто спрашивать, разрешима ли вообще задача, теория сложности спрашивает, насколько трудно решить эту задачу. Другими словами, теория сложности интересуется возможностями универсальных вычислительных устройств, таких, как машина Тьюринга, при наличии ограничений на время выполнения или объем памяти, которая может быть использована. Первые проблески теории сложности появились в статьях, опубликованных в 1959 и 1960 гг. Микаэлем Рабином, а также Робертом Макнотомом и Хидео Ямадой, но начало современной эры теории сложности отметила статья 1965 г. Юриса Хартманиса и Ричарда Стирнса. Используя машину Тьюринга как модель абстрактного вычислителя, Хартманис и Стирнс предложили точное определение «класса сложности», состоящего из всех задач, решаемых за число шагов, ограниченных некоторой данной функцией длины входа  $n$ . Приспосабливая диагональный метод, который Тьюринг использовал для доказательства неразрешимости проблемы остановки, они доказали много интересных результатов о структуре классов сложности. Все, кто прочитал их статью, не могли не понять, что теперь имеется удовлетворительная формальная структура для работы над вопросами, которые Эдмондс поставил ранее интуитивно, — вопросами о том, например, разрешима ли задача о коммивояжере за полиномиальное время.

В том же самом году я изучал теорию вычислимости по великолепной книге Хартли Роджерса, который ранее был моим учителем в Гарварде. Я вспоминаю, как спрашивал себя тогда, может ли понятие сводимости, которое занимает столь важное место в теории вычислимости, играть роль в теории сложности, но я не видел, как осуществить эту связь. Примерно в то же время Микаэль Рабин, который впоследствии получил тьюринговскую премию в 1976 г., был гостем Исследования

тельской лаборатории IBM в Йорктаун Хайтс, приехав из Еврейского университета в Иерусалиме. Нам обоим случилось жить в одном и том же доме на окраине Нью-Йорка, и мы завели привычку совершать вместе поездку к Йорктаун Хайтс и обратно. Рабин — глубоко оригинальный мыслитель и один из основателей как теории автоматов, так и теории сложности, и благодаря своим ежедневным дискуссиям с ним во время поездок вдоль Соумилл ривер парквэй я получил гораздо более широкий взгляд на логику, теорию вычислимости и теорию абстрактных вычислительных машин.

В 1968 г., — возможно, под влиянием общественных волнений, охвативших страну, — я решил перебраться в Калифорнийский университет в Беркли, который был центром студенческого движения. Годы в IBM были решающими в моем становлении как ученого. Возможность работать с такими выдающимися учеными, как Алан Хофман, Раймонд Миллер, Арнольд Розенберг и Шмуэль Виноград, была просто бесценной. Мой новый круг коллег включал Майкла Харрисона, известного специалиста по теории языков, который и соблазнил меня перебраться в Беркли; Юджина Лоулера, эксперта по комбинаторной оптимизации; Мануэля Блюма, основателя теории сложности, который потом выполнил выдающуюся работу на границе теории чисел и криптографии, и Стивена Кука, чья работа по теории сложности повлияет на меня так сильно через несколько лет. В Отделении математики работали Джулия Робинсон, чья работа над 10-й проблемой Гильберта вскоре принесет плоды; Роберт Соловей, знаменитый логик, открывший важный рандомизированный алгоритм распознавания простоты числа, и Стив Смейл, чья потрясающая работа по вероятностному анализу алгоритмов линейного программирования оказала на меня влияние через несколько лет. А на другом берегу Калифорнийской бухты, в Станфорде, были Данциг, отец линейного программирования, Дональд Кнут, основавший область структур данных и анализа алгоритмов, а также Роберт Тарьян, тогда студент старшего курса, и Джон Хопкрофт, приглашенный профессор из Корнеллского университета, который блестяще применял технику структур данных к анализу графовых алгоритмов.

В 1971 г. Кук, который к тому времени переехал в Университет Торонто, опубликовал свою историческую статью «О сложности процедур доказательства теорем». Кук рассмотрел классы задач, которые мы теперь называем  $P$  и  $NP$ , и ввел понятие, которое мы теперь называем  $NP$ -полнотой. Неформально класс  $P$  состоит из всех тех задач, что могут быть решены за полиномиальное время. Так, задача марьяжа лежит в  $P$ , потому что венгерский алгоритм решает частный случай раз-

мера  $n$  примерно за  $n^3$  шагов, но задача о коммивояжере, похоже, не лежит в  $P$ , так как каждый из известных методов ее решения требует экспоненциального времени. Если исходить из того, что вычислительная задача практически неразрешима, если не существует алгоритма с полиномиальным временем для ее решения, то все практически решаемые задачи лежат в  $P$ . Класс  $NP$  состоит из всех тех задач, для которых любое предложенное решение может быть проверено за полиномиальное время. Например, рассмотрим версию задачи о коммивояжере, в которой входные данные состоят из расстояний между всеми парами городов и «целевого числа»  $T$ , и задание состоит в определении того, существует ли маршрут длины, меньшей или равной  $T$ . По-видимому, крайне трудно определить, существует ли такой маршрут, но если такой маршрут нам задан, мы можем легко проверить, верно ли, что его длина меньше или равна  $T$ ; следовательно, эта версия задачи о коммивояжере лежит в классе  $NP$ . Подобным образом через введение целевого числа  $T$  все комбинаторные задачи оптимизации, обычно рассматриваемые в областях коммерции, науки и техники, имеют версии, лежащие в классе  $NP$ .

Таким образом,  $NP$  представляет собой область, в которую обычно попадают комбинаторные задачи; внутри  $NP$  лежит  $P$ , класс задач, имеющих эффективные решения. Фундаментальный вопрос состоит в следующем: как соотносятся класс  $P$  и класс  $NP$ ? Ясно, что  $P$  является подклассом  $NP$ , и вопрос, к которому Кук привлек внимание, состоит в том, могут ли  $P$  и  $NP$  быть одним и тем же классом. Если бы  $P$  совпадал с  $NP$ , то возникли бы ошеломляющие следствия: это означало бы, что всякая задача, решения для которой легко проверить, решались бы легко; это означало бы, что если теорема имеет короткое доказательство, то некой универсальной процедурой можно было бы найти это доказательство быстро; это означало бы, что все обычные задачи комбинаторной оптимизации решались бы за полиномиальное время. Короче, это означало бы, что можно избавиться от проклятия комбинаторных взрывов. Но несмотря на все эти эвристические доводы о том, что было бы слишком хорошо, если бы  $P$  и  $NP$  совпадали, никакого доказательства, что  $P \neq NP$ , найдено не было, и некоторые специалисты даже верят в то, что никакого доказательства никогда и не будет найдено.

Наиболее важное достижение статьи Кука состоит в доказательстве того, что  $P = NP$  тогда и только тогда, когда конкретная вычислительная задача, называемая проблемой выполнимости, принадлежит  $P$ . Проблема выполнимости происходит из математической логики и имеет приложения в теории переключательных схем, но может быть сформулирована как простая:

комбинаторная головоломка: можно ли из нескольких заданных последовательностей прописных и строчных букв выбрать по букве из каждой последовательности, не выбирая обе (прописную и строчную) версии никакой буквы? Например, если последовательности суть  $Abc$ ,  $BC$ ,  $aB$  и  $ac$ , можно выбрать  $A$  из первой последовательности,  $B$  из второй и третьей и  $c$  из четвертой; отметим, что одна и та же буква может быть выбрана более чем однажды при условии, что мы не выбираем обе ее (прописную и строчную) версии. Пример, где требуемый выбор невозможен, задается следующими четырьмя последовательностями:  $AB$ ,  $Ab$ ,  $aB$  и  $ab$ .

Проблема выполнимости, очевидно, принадлежит  $NP$ , так как легко проверить, удовлетворяет ли условиям задачи предложенный выбор букв. Кук доказал, что если проблема выполнимости решается за полиномиальное время, то всякая задача из  $NP$  тоже решается за полиномиальное время, так что  $P=NP$ . Таким образом, мы видим, что эта кажущаяся странной и непоследовательной задача есть архетипическая комбинаторная задача, она служит ключом к эффективному решению всех задач из  $NP$ .

Доказательство Кука было основано на понятии сводимости, которое мы упоминали ранее, когда говорили о теории вычислимости. Он показал, что всякий частный случай задачи из  $NP$  может быть преобразован в соответствующий случай проблемы выполнимости таким образом, что оба они одновременно или имеют, или не имеют решения. Более того, это преобразование может быть выполнено за полиномиальное время. Другими словами, проблема выполнимости является достаточно общей для фиксации структуры любой задачи из  $NP$ . Отсюда следует, что если бы мы умели решить проблему выполнимости за полиномиальное время, то мы сумели бы построить алгоритм, работающий за полиномиальное время, для любой задачи из  $NP$ . Этот алгоритм состоял бы из двух частей: процедуры, переводящей за полиномиальное время частные случаи данной задачи в частные случаи проблемы выполнимости, и процедуры для решения самой проблемы выполнимости за полиномиальное время (рис. 6).

По прочтении статьи Кука я тотчас же понял, что его концепция архетипической комбинаторной задачи была формализацией идеи, уже давно бывшей частью фольклора комбинаторной оптимизации. Работающие в этой области понимали, что задача целочисленного программирования, которая по существу есть проблема разрешимости для системы линейных неравенств в целых числах, является достаточно общей, чтобы выразить условия любой из обычно встречающихся задач комбинаторной оптимизации. Данциг опубликовал статью на эту тему в 1960 г.

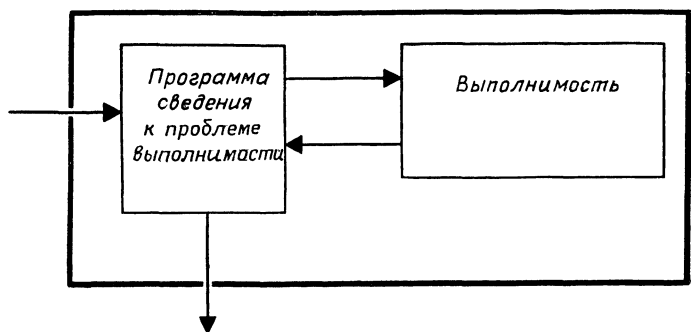


Рис. 6. Задача о коммивояжере за полиномиальное время сводима к проблеме выполнимости.

Из-за того что Кук интересовался доказательством теорем, а не комбинаторной оптимизацией, он выбрал другую архетипическую задачу, но основная идея была та же самая. Однако было решающее отличие: используя аппарат теории сложности, Кук построил формализм, в котором архетипическая природа данной задачи смогла стать теоремой, а не неформальным тезисом. Интересно, что Леонид Левин, который тогда работал в Ленинграде, а сейчас профессор Бостонского университета, независимо открыл по существу ту же самую совокупность идей. Его архетипическая задача была связана с покрытием конечных областей плоскости костями домино.

Я решил исследовать, являются ли некоторые классические комбинаторные задачи, которые издавна считались практически невыполнимыми, архетипическими в смысле Кука. Я назвал такие задачи «полиномиально полными», но этот термин был вытеснен более точным термином «NP-полные». Задача NP-полна, если она лежит в классе NP и всякая задача из NP полиномиально сводима к ней. Таким образом, по теореме Кука проблема выполнимости NP-полна. Для доказательства того, что задача из NP является NP-полной, достаточно показать, что некоторая задача, NP-полнота которой доказана, полиномиально сводима к данной задаче. Построением серии полиномиальных по времени сведений я показал, что большинство классических задач упаковки, покрытия, паросочетания, разбиения, выбора маршрутов и составления расписаний, которые возникают в комбинаторной оптимизации, NP-полны. Я представил эти результаты в 1972 г. в статье под названием «Взаимная сводимость комбинаторных задач». Мои ранние результаты были быстро уточнены и усилены другими авторами, и в следующие несколько лет для сотен различных задач, возникающих фактически

в каждой области, где делаются вычисления, было показано, что они NP-полны.

## РАБОТА С NP-ПОЛНЫМИ ЗАДАЧАМИ

Я был вознагражден за мои достижения в исследовании NP-полных задач административным постом. С 1973 по 1975 г. я возглавлял вновь образованный отдел информатики в Беркли, и мои обязанности оставляли мне мало времени для исследований. В результате я оказался на обочине в самый активный период, в то время, когда были найдены многие примеры NP-полноты и были предприняты первые попытки обойти отрицательные следствия NP-полноты.

Результаты по NP-полноте, доказанные в начале 1970-х годов, показали, что огромное большинство задач комбинаторной оптимизации, возникающих в коммерции, науке и технике, практически невыполнимы, если не верно  $P=NP$ ; никакие методы их решения не могут полностью избежать комбинаторных взрывов. Как же тогда нам справляться с такими задачами на практике? Один из возможных подходов исходит из того, что достаточно хороши решения, близкие к оптимальным: коммивояжер, вероятно, будет удовлетворен маршрутом, на несколько процентов длиннее оптимального. Следуя такому подходу, исследователи начали искать полиномиальные по времени алгоритмы, которые гарантируют почти оптимальные решения NP-полных комбинаторных задач. В большинстве случаев критерий эффективности приближенного алгоритма формулировался в виде верхней оценки отношения стоимости решения этим алгоритмом к стоимости оптимального решения.

Некоторые из наиболее интересных работ по приближенным алгоритмам с гарантиями эффективности содержали одномерную задачу упаковки в контейнер. В этой задаче совокупность объектов различных размеров должна быть упакована в контейнеры одинаковой емкости. Цель состоит в минимизации числа контейнеров, используемых для упаковки, при условии, что сумма размеров упаковываемых объектов в каждом контейнере не должна превосходить емкости контейнера. В середине 1970-х серия статей по приближенным алгоритмам упаковки в контейнеры достигла кульминации в анализе Дэвида Джонсона алгоритма «первый подходящий по убыванию». В этом простом алгоритме объекты рассматриваются в порядке убывания их размеров, и каждый объект — по очереди — помещается в первый из контейнеров, который может его принять. В примере, изображенном на рис. 7, например, есть четыре контейнера, емкость каждого равна 10, и девять предметов по размеру в пределах от 2 до 8. Джонсон показал, что этот простой ме-

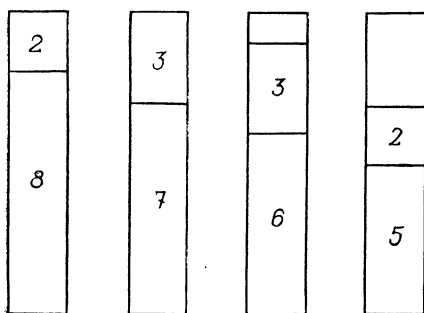


Рис. 7. Упаковка, порождаемая алгоритмом «первый подходящий по убыванию».

тод гарантирует относительную погрешность, не превышающую  $2/9$ , другими словами, число требуемых контейнеров никогда не превосходит больше чем примерно на 22 процента числа контейнеров в оптимальном решении. Через несколько лет эти результаты были еще улучшены, и в конце концов было показано, что относительная погрешность может быть сделана сколь угодно малой, хотя в этом полиномиальном по времени алгоритме

отсутствовала простота проанализированного Джонсоном алгоритма «первый подходящий по убыванию».

Исследования полиномиальных по времени приближенных алгоритмов выявили интересные различия среди NP-полных задач дискретной оптимизации. Для некоторых задач относительная погрешность может быть сделана сколь угодно малой, для других ее можно снизить до некоторого уровня, но, по-видимому, не более того, иные задачи сопротивляются всем попыткам найти алгоритмы с ограниченной относительной погрешностью, и, наконец, существуют задачи, для которых существование полиномиального по времени приближенного алгоритма с ограниченной относительной погрешностью означало бы, что  $P=NP$ .

В течение седьмого года, который последовал за моим пребыванием на административном посту, я начал размышлять о разрыве между теорией и практикой в области комбинаторной оптимизации. В области теоретической новости были неутешительны. Почти все задачи, которые хотелось решить, были NP-полны, и в большинстве случаев полиномиальные по времени алгоритмы не могли дать таких гарантий эффективности, которые были бы полезны на практике. Тем не менее было много алгоритмов, которые, судя по всему, работают вполне хорошо на практике, хотя они и нуждаются в теоретическом обосновании. Например, Линь и Керниган разработали весьма успешную стратегию локальных улучшений для задачи о коммивояжере. Их алгоритм просто стартовал со случайного маршрута и все улучшал и улучшал его добавлением и исключением нескольких ребер, пока маршрут в конце концов не становился таким, что не допускал локальных улучшений. В специально построенных частных случаях их алгоритм работал ужасно, но на практике он приводил к почти оптимальным решениям.

Подобная ситуация наблюдается и для симплекс-алгоритма, одного из наиболее важных среди всех вычислительных методов: он надежно решает те крупные задачи линейного программирования, которые возникают в приложениях, несмотря на то, что в некоторых искусственно построенных примерах он выполнялся за экспоненциальное число шагов.

Представлялось, что успех таких неточных или теоретически необоснованных алгоритмов был эмпирическим фактом, требующим объяснения. Кроме того, представлялось, что объяснение этого факта будет неизбежно требовать отхода от традиционных парадигм теории сложности, которые оценивают алгоритм согласно его эффективности в худшем из возможных случаев. Традиционный анализ худшего случая — доминирующее направление теории сложности — соответствует сценарию, в котором частные случаи решаемой задачи построены бесконечно умным противником, который знает структуру алгоритма и подбирает входную информацию так, что будет предельно усложнять работу этого алгоритма. Такой сценарий приводит к заключению, что симплекс-алгоритм и алгоритм Линя — Кернигана безнадежно дефектны. Я начал изучать другой подход, в котором входная информация предполагается поступающей от пользователя, который просто выбирает свои частные случаи, исходя из некоторого разумного распределения вероятности, не пытаясь ни повредить, ни помочь алгоритму.

В 1975 г. я решил попытаться счастья и взялся за исследование вероятностного анализа комбинаторных алгоритмов. Я должен сказать, что это решение потребовало некоторой храбрости, так как имело своих оппонентов, которые совершенно правильно указывали, что нет никакого способа узнать, какая именно входная информация поступит на входы алгоритма, и что лучший вид гарантий, если можно получить, их, — гарантии для худшего случая. Я чувствовал, однако, что в случае NP-полных задач мы не получим желаемых гарантий для худшего случая и что вероятностный подход — лучший путь — и, возможно, единственный — к пониманию того, почему эвристические комбинаторные алгоритмы работают так хорошо на практике.

Вероятностный анализ начинается с предположения, что частные случаи задачи представляют собой репрезентативную выборку из генеральной совокупности с заданным распределением вероятностей. В случае задачи о коммивояжере, например, одно из возможных допущений состоит в том, что расположение каждого из  $n$  городов выбирается независимо из равномерного распределения на единичном квадрате. В этом предположении мы можем изучать распределение вероятностей длины оптимального маршрута или длину маршрута, построенного



тем или иным алгоритмом. В идеале цель состоит в доказательстве того, что некоторый простой алгоритм дает оптимальные или почти оптимальные решения с высокой вероятностью. Конечно, такой результат осмыслен только в том случае, когда распределение вероятностей для частных случаев задачи некоторым образом подобно совокупности случаев, возникающих в реальной жизни, или тогда, когда вероятностный анализ достаточно нечувствителен к характеристикам распределения, чтобы быть пригодным для широкого диапазона возможных распределений вероятностей.

Один из наиболее удивительных феноменов теории вероятностей — закон больших чисел, согласно которому кумулятивный эффект большого числа случайных событий в высокой степени предсказуем, даже если исходы индивидуальных событий совершенно непредсказуемы. Например, мы можем уверенно предсказать, что в длинной серии бросаний монеты примерно половина исходов — орел. Вероятностный анализ обнаружил, что то же явление управляет поведением многих комбинаторных алгоритмов оптимизации, когда входная информация подчинена простому распределению вероятностей: с очень высокой вероятностью выполнение алгоритма развивается хорошо предсказуемым образом и полученное решение близко к оптимальному. Например, статья 1960 г. Бердвуда, Холтона и Хаммерсли показывает, что если  $n$  городов в задаче о коммивояжере выбираются независимо с равномерным распределением на единичном квадрате, то при достаточно большом  $n$  длина оптимального маршрута будет почти наверняка очень близкой к некоторой универсальной константе, умноженной на корень квадратный из числа городов. Воодушевленный этим результатом, я показал, что, когда число городов чрезвычайно велико, простой алгоритм «разделяй и властвуй» почти всегда будет приводить к маршруту с длиной, очень близкой к длине оптимального маршрута (рис. 8). Алгоритм стартует с разбиения области, где расположены города, на прямоугольники, каждый из которых содержит малое число городов. Затем он строит оптимальный маршрут для городов каждого прямоугольника. Объединение всех этих маленьких маршрутов сильно напоминает общий маршрут, но отличающийся от него лишними посещениями пограничных городов. Наконец, алгоритм выполняет своего рода «локальную хирургию» с целью исключения этих излишних посещений и образования маршрута.

Можно привести еще много примеров, где простые приближенные алгоритмы почти наверняка дают близкие к оптимальным решения для случайных «больших» частных случаев NP-полных оптимизационных задач. Например, моя студентка Салли Флloyd, продолжая ранние работы об упаковке в контей-

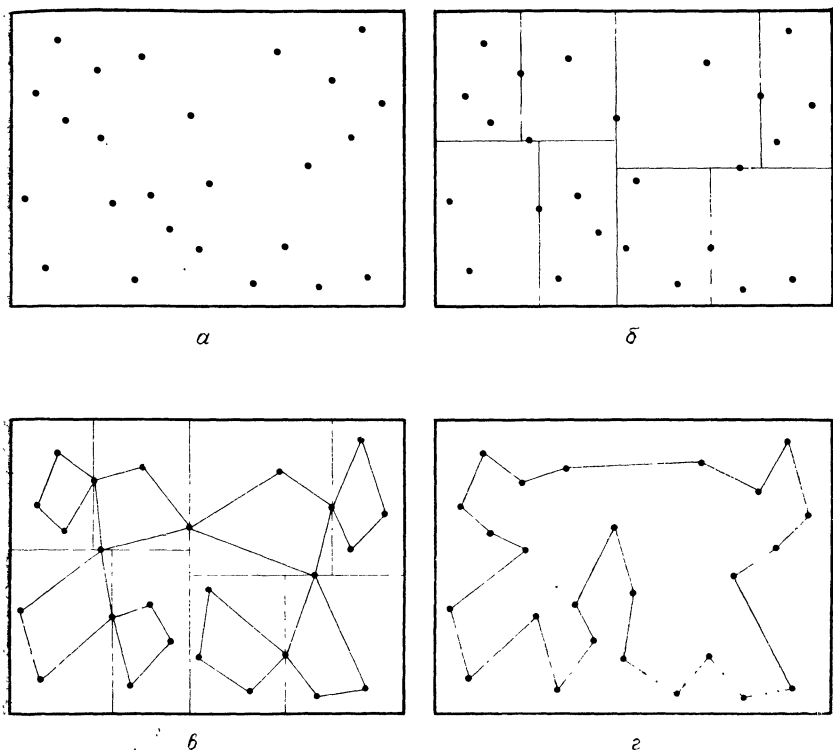


Рис. 8. Алгоритм «разделяй и властвуй» для задачи о коммивояжере на плоскости.

неры Бентли, Джонсона, Лейтона, Макгеоха и Макгеоха, показала, что если объекты, которые нужно упаковать, выбираются независимо с равномерным распределением на интервале  $[0, 1/2]$ , то независимо от числа объектов алгоритм «первый подходящий по убыванию» почти наверняка даст упаковку, в которой неиспользованный объем не превышает объема 10 контейнеров.

Среди наиболее замечательных приложений вероятностного анализа — его применение к задаче линейного программирования. Геометрически эта задача сводится к нахождению вершины многогранника, ближайшей к некоторой внешней гиперплоскости. Алгебраически это эквивалентно минимизации линейной функции в области, ограниченной линейными неравенствами. Линейная функция измеряет расстояние до гиперплоскости, а линейные неравенства соответствуют гиперплоскостям, ограничивающим многогранник.

Симплекс-алгоритм для задач линейного программирования есть метод взбирания на гору. Он все время переходит от одной вершины к соседней вершине, всегда приближаясь к внешней гиперплоскости. Алгоритм заканчивает работу, когда он достигает вершины, ближайшей к этой гиперплоскости по сравнению с соседними вершинами; эта вершина заведомо есть оптимальное решение. В худшем случае симплекс-алгоритм требует растущего экспоненциально числа операций относительно числа линейных неравенств, необходимых для описания многогранника, но на практике число итераций редко превосходит число линейных неравенств более чем в три или четыре раза.

Карл-Хейнц Боргвардт из Западной Германии и Стив Смэйл из Беркли были первыми исследователями, использовавшими вероятностный анализ для объяснения непонятого успеха симплекс-алгоритма и его вариаций. Их анализ основан на вычислении некоторых многомерных интегралов. С моей ограниченной подготовкой в области математического анализа я нашел их метод трудновоспринимаемым. К счастью, один из моих коллег в Беркли, Айлан Адлер, предложил один подход, который сулил привести к вероятностному анализу вовсе без вычислений. При этом используются некоторые принципы симметрии, чтобы сделать требуемые усреднения и волшебным образом прийти к ответу.

Следуя этому направлению исследований, Адлер, Рон Шамир и я показали в 1983 г., что при весьма широких допущениях о вероятностных характеристиках входных данных ожидаемое число итераций, выполненных одной из версий симплекс-алгоритма, растет всего лишь как квадрат числа линейных неравенств. Тот же самый результат получил при помощи многомерных интегралов Майкл Тодд, а также Адлер и Нимрод Меджиддо. Я считаю, что эти результаты существенно помогают понять, почему симплекс-метод работает так хорошо.

Вероятностный анализ алгоритмов комбинаторной оптимизации был главной темой моих исследований за последнее десятилетие. В 1975 г., когда я впервые занялся такими исследованиями, было очень мало примеров анализа этого рода. К настоящему времени на эту тему написаны сотни статей, и все классические комбинаторные задачи были подвергнуты вероятностному анализу. Эти результаты привели к существенному пониманию того, насколько эти задачи могут быть «приручены» на практике. Тем не менее я думаю, что это предприятие удалось лишь отчасти. В силу ограничений существующих методов мы продолжаем работать с простейшими вероятностными моделями, и даже тогда многие из наиболее интересных и успешных алгоритмов не поддаются анализу. После всего, что было

сказано и сделано, построение практических алгоритмов комбинаторной оптимизации остается в той же мере искусством, что и наукой.

## РАНДОМИЗИРОВАННЫЕ АЛГОРИТМЫ

Алгоритмы, в процессе выполнения которых бросается монета, время от времени предлагались с тех пор, как появились первые компьютеры, но систематическое изучение таких рандомизированных алгоритмов началось только примерно с 1976 г. Интерес к этой теме привлекли два удивительно эффективных рандомизированных алгоритма проверки того, является ли число  $n$  простым; один из этих алгоритмов предложили Соловей и Фолкер Штрассен, а другой — Рабин. Следующая статья Рабина дала дальнейшие примеры и обоснование систематического изучения рандомизированных алгоритмов, и докторская диссертация Джона Гилла под руководством моего коллеги Блюма заложила основы общей теории рандомизированных алгоритмов.

Чтобы понять преимущества бросания монеты, вернемся снова к сценарию, связанному с анализом худшего случая, в котором всезнающий противник выбирает те частные случаи, в которых алгоритм работает хуже всего. Рандомизация делает поведение алгоритма непредсказуемым, даже когда частный случай фиксирован, и так можно сделать трудным и даже невозможным для противника выбрать частный случай, который затруднит работу алгоритма. Существует полезная аналогия с футболом, в которой алгоритм подобен наступающей команде и противник защищается. Детерминированный алгоритм подобен команде, которая полностью предсказуема в тактике игры, что позволяет другой команде построить непробиваемую защиту. Как знает всякий полузащитник, внесение небольшого разнообразия в игру достаточно, чтобы заставить защищающую команду играть честно.

В качестве конкретной иллюстрации преимуществ бросания монеты я приведу простой рандомизированный алгоритм сопоставления с образцом, изобретенный Рабином и мною в 1980 г. Задача сопоставления с образцом — одна из основных в обработке текстов. Задана цепочка  $n$  бит, называемая образцом, и гораздо более длинная цепочка, называемая текстом; задача состоит в том, чтобы обнаружить, входит ли образец в текст как связный блок (рис. 9). Грубый метод решения этой задачи состоит в непосредственном сравнении образца с каждым  $n$ -рядным блоком внутри текста. В худшем случае время выполнения этого метода пропорционально произведению длины образца на длину текста. Во многих приложениях обработки

Образец 11001

Текст 011011101 11001 00

Рис. 9. Проблема сопоставления с образцом.

текстов этот метод работает неприемлемо медленно, кроме случая очень коротких образцов.

Наш метод обходит эту трудность при помощи приема простого вырубания. Мы определим «функцию отпечатков пальцев», которая связывает с каждой  $n$ -разрядной последовательностью гораздо более короткую, называемую «отпечатком пальца». Функция отпечатка выбрана так, что можно пройти по тексту, быстро вычисляя отпечаток каждого  $n$ -разрядного блока. Тогда вместо того, чтобы сравнивать образец с каждым таким блоком текста, мы сравниваем отпечаток образца с отпечатком каждого такого блока. Если отпечаток образца отличается от отпечатков всех блоков, то мы знаем, что образец не появляется в качестве блока в тексте.

Метод сравнения коротких отпечатков вместо длинных цепочек сильно уменьшает время работы, но приводит к возможности ложных сопоставлений, когда образец и некоторый блок текста имеют одинаковые отпечатки, хотя сами они не совпадают. Ложное сопоставление представляет серьезную проблему: фактически для всякого конкретного выбора функции отпечатков противник может построить такой пример образца и текста, что ложное сопоставление возникнет в каждой позиции текста. Поэтому необходимо некоторое дублирование для защиты от ложных сопоставлений, и преимущества метода отпечатков, казалось бы, должны потеряться.

К счастью, преимущества метода отпечатков можно сохранить при помощи рандомизации. Вместо работы с единственной функцией отпечатков рандомизированный метод имеет в своем распоряжении большое семейство различных легковычислимых функций отпечатков. Когда частный случай, состоящий из образца и текста, предъявлен, алгоритм наудачу выбирает функцию отпечатка из этого большого семейства и использует эту функцию для проверки совпадения образца и текста. Из-за того что функция отпечатков заранее неизвестна, для противника невозможно построить частный случай задачи, который приводит к ложному сопоставлению. Можно показать, что независимо от выбора образца и текста вероятность ложного сопоставления очень мала. Например, если длина образца составляет 250 бит и текста — 4000 бит, можно работать с легковычислимыми 32-битовыми отпечатками, и все еще будет гарантия, что вероятность ложного сопоставления меньше одной

тысячной для каждого возможного частного случая. Во многих приложениях обработки текстов эта вероятностная гарантия достаточно хороша, чтобы исключить необходимость повторной проверки, и таким образом преимущества метода отпечатков снова восстановлены.

Рандомизированные алгоритмы и вероятностный анализ алгоритмов — два разных способа избежать анализа эффективности детерминированных алгоритмов в худшем случае. В первом способе случайность вводится в поведение самого алгоритма, а во втором предполагается, что случайность присутствует в выборе частных случаев. Подход, основанный на рандомизированных алгоритмах, конечно, более привлекателен из этих двух, так как он обходится без предположений о среде, в которой алгоритм будет использоваться. Однако пока не показана эффективность рандомизированных алгоритмов в борьбе с комбинаторными взрывами, характерными для NP-полных задач, и, видимо, оба этих подхода будут иметь свои применения.

## ЗАКЛЮЧЕНИЕ

Итак, мой рассказ подошел к концу, и мне бы хотелось закончить кратким замечанием, что значит сегодня работать в теоретической информатике. Когда я принимаю участие в ежегодном симпозиуме АСМ по теории вычислений, или присутствую на ежемесячном теоретическом семинаре Залива, или поднимаюсь на холм позади университетского городка Беркли к Математическому научно-исследовательскому институту, где выполняется годовая программа по вычислительной сложности, — я поражен масштабами работы, проводимой в этой области. Я горд, что связан с областью исследования, в которой выполнено так много великолепных работ, и рад, что я в состоянии время от времени помогать очень талантливым молодым исследователям стать на ноги в этой области. Благодарю вас за предоставленную мне возможность выступить как представителю моей области на этом собрании.

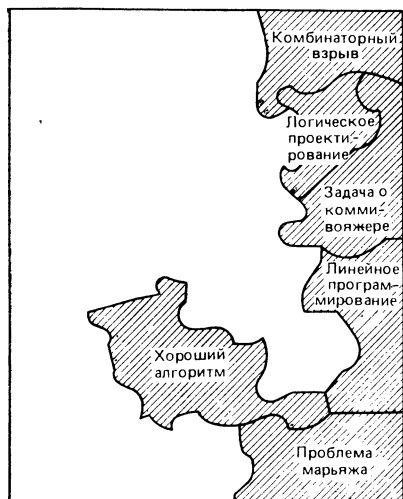
# Постскриптум

## Сборка теории сложности из кусков

Карен А. Френкель

Обозреватель Communications of the ACM

Для иллюстрации того, «в какой удивительной степени в теории сложности употребляются аналогии теории вычислимости», Ричард Карп построил эту концептуальную карту или головоломку. Чтобы собрать эту головоломку на плоскости, он использует «алгоритм для плоских графов». Наиболее удаленные друг от друга части могут при первом рассмотрении казаться независимыми, «но в конце концов теория NP-полноты собирает их вместе», говорит Карп.



В верхнем правом углу головоломки располагаются понятия, относящиеся к комбинаторным взрывам, и понятия «хорошего» или «эффективного» алгоритма. В свою очередь «слож-

ность» связывает эти понятия с левым верхним углом, в котором представлено то, что интересовало прежних специалистов по теории вычислимости.

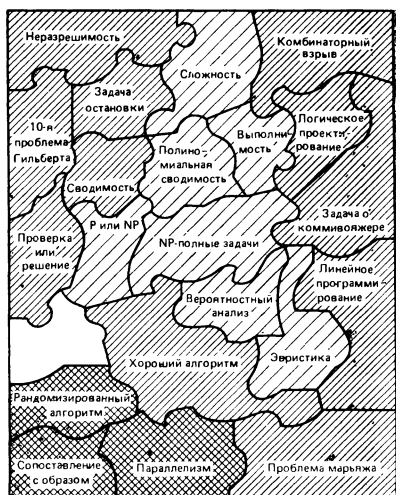
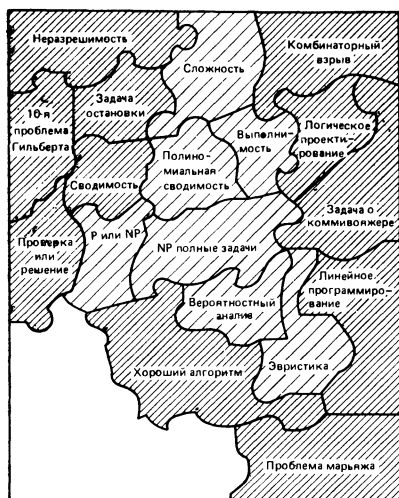
Задача о коммивояжере ближе к верхнему правому углу, потому что она, по всей вероятности, практически неразрешима. Она поэтому граничит с «*NP*-полнотой» и «Комбинаторным взрывом».

До некоторой степени, однако, некоторые границы нечетки. «Линейное программирование», например, имеет аномальный статус — наиболее широко используемые на практике алгоритмы решения таких задач нехороши в теоретическом смысле, а те, что хороши теоретически, нехороши для практики. Одним из примеров является метод эллипсоидов, бывший объектом пристального внимания шесть лет назад. Он выполнялся за полиномиальное время, но полином был такой высокой степени, что этот метод хорош лишь в смысле формального соответствия теоретическому критерию и неэффективен на практике. «Причина состоит в том, что понятие полиномиального по времени алгоритма неточно передает содержание интуитивного представления об эффективных алгоритмах», — объясняет Карп. — Когда степень полинома доходит до 5 или 6, то трудно назвать такой алгоритм действительно эффективным. Таким образом, предложенное Эдмондсом понятие хорошего алгоритма — недостаточно совершенная формализация хорошего в интуитивном смысле». Далее, симплекс-алгоритм хорош на практике во всех отношениях, замечает Карп, но недостаточно хорош в соответствии со стандартной парадигмой теории сложности. Последнее добавление к решениям линейного программирования — алгоритм, построенный Нарендой Кармаркаром и, по мнению некоторых, бросающий вызов симплекс-алгоритму, хорош в техническом смысле и также, видимо, вполне эффективен на практике, говорит Карп.

«Хороший алгоритм» примыкает к «Эвристике», потому что эвристический алгоритм может работать хорошо, но имеет недостаточное теоретическое обоснование. Некоторые эвристические алгоритмы всегда быстры, но иногда не могут дать хороших решений. Другие всегда дают оптимальные решения, но не гарантируют быстроты выполнения. Симплекс-алгоритм — последнего типа.

«Неразрешимость», «Комбинаторный взрыв» и «Сложность» располагаются на одном уровне, потому что они аналогичны друг другу; неразрешимость включает неограниченный поиск, тогда как комбинаторные взрывы по определению представляют собой очень длинные, но не неограниченные поиски. Теория сложности заполняет разрыв, потому что вместо того, чтобы спрашивать о том, может ли быть вообще решена задача,





она ставит вопросы о *ресурсах*, необходимых для решения задачи.

В нижнем левом углу содержатся сегменты, которые интересовали Карпа в самое последнее время и в которых имеются открытые вопросы. «Рандомизированный алгоритм», например, располагается напротив «Вероятностного анализа», потому что это две альтернативы анализа детерминированных алгоритмов: в худшем случае. Рандомизированные алгоритмы могут быть способны решать задачи за полиномиальное время, тогда как детерминированные не могут, и это могло бы означать расширение понятия хороших алгоритмов. Возможно, через построение программных конструкций не фон-неймановских машин можно построить более эффективные на практике алгоритмы посредством параллелизма.

Наконец, некоторые части головоломки еще не определены. Как говорит Карп, «они соответствуют неизвестной теории, которую предстоит исследовать в будущем».

# Постскрипtum

## Интервью тьюринговских лауреатов. Сложность и параллельные вычисления

ИНТЕРВЬЮ С РИЧАРДОМ КАРПОМ

*Карен А. Френкель*

Обозреватель Communications of the ACM

В приводимом ниже интервью, которое он дал на конференции АСМ 1985 г. в Денвере, Карп обсуждает связь между его работами и новейшими вычислительными дисциплинами, такими, как параллельные вычисления и искусственный интеллект. Проследившая свой опыт новатора в высокой степени теоретических исследованиях в области информатики, Карп описывает, как решение пойти против устоявшихся взглядов привело к работе, благодаря которой он наиболее известен, и как находки коллег позволили ему увидеть связи между этими двумя прежде независимыми областями. Подчеркивается важность обмена идеями с коллегами, который помогает найти новые ключевые принципы.

**К. Ф.** Вы решили перейти от математики к информатике в самом начале вашей карьеры. Ощущаете ли вы себя теоретическим математиком, работающим в области информатики, или специалистом по информатике, работающим над ее теоретическими аспектами?

**Р.К.** Я думаю, что я представляю собой нечто среднее между прикладным математиком и специалистом по информатике. По-моему, а priori мою работу можно вести и на факультете математики и на факультете информатики, но исторически сложилось так, что серьезные начинания в развитии теоретической информатики брали на себя факультеты информатики.

Большинство математических факультетов «упустили мяч». Есть исключения, но, как правило, они не поняли вовремя потенциал этой области и не начали ее развивать. Таким образом, она попадала в зону внимания факультетов информатики. Сейчас математические факультеты наконец-то все более признают теоретическую информатику.

**К.Ф.** Математики и специалисты по информатике по-разному подходят к вычислениям?

**Р.К.** Когда математики применяют ЭВМ, они склонны действовать совсем не как теоретики. Если специалисту по теории чисел нужно разложить на множители какое-то число, он не-

пожалеем для этого усилий. Ему нужен ответ, а более общие вопросы сложности вычислений его не волнуют. Так же обстоит дело с работающими в теории групп или в алгебраической геометрии. Их интересует конкретная группа или конкретная поверхность, и для них им нужен ответ — они становятся совсем как инженеры. И я такой же, когда программирую. Первые пять минут не забываю о вопросах теории, а потом мне просто нужно, чтобы программа работала, и я забываю, что я теоретик.

**К. Ф. Почему такое большое внимание уделяется задаче о коммивояжере?**

**Р.К.** Задача о коммивояжере — это прототип и упрощенный вариант встречающихся в практике более трудных задач. Все знают, что задача о коммивояжере — это метафора или миф; ясно, что ни один коммивояжер не будет стремиться к абсолютной минимизации своего километража. Но это задача интересная и просто формулируемая. Вероятно, она привлекает внимание в большей степени, чем того заслуживает, из-за своего завлекательного названия. Есть и другие важные задачи-прототипы с не столь завлекательными названиями, такие, как раскраска, упаковка, паросочетание, составление расписания и т. д. Так и развивается теория — если учитывать все сложности реального мира, то чистой теорией заниматься нельзя. Поэтому упрощаешь постановки задач, изучаешь их возможно тщательнее, глубоко вникаешь в их структуру и надеешься, что результаты будут переноситься на реальные проблемы.

**К. Ф. Похоже, вы исследуете метатеорию — классы задач, а не реальные задачи.**

**Р.К.** Да, верно. Есть три уровня проблем. Есть уровень решения вполне конкретного случая задачи: скажем, вам нужен кратчайший маршрут по 48 столицам континентальных штатов плюс Вашингтон. К практике этот уровень ближе всего. Затем есть уровень изучения общей задачи с упором на методы ее решения: вы хотите знать, какова сложность задачи о коммивояжере или задачи о подборе пар для худшего случая. Этот уровень выше, поскольку вас интересует не просто конкретный случай. Затем есть уровень метатеории, на котором изучается вся структура целого класса задач. Такую точку зрения мы унаследовали у логики и теории вычислимости. Вопрос приобретает вид: «Какова в целом схема классификации? Давайте взглянем на иерархию сложностей задач по мере возрастания предельно допустимого времени вычисления».

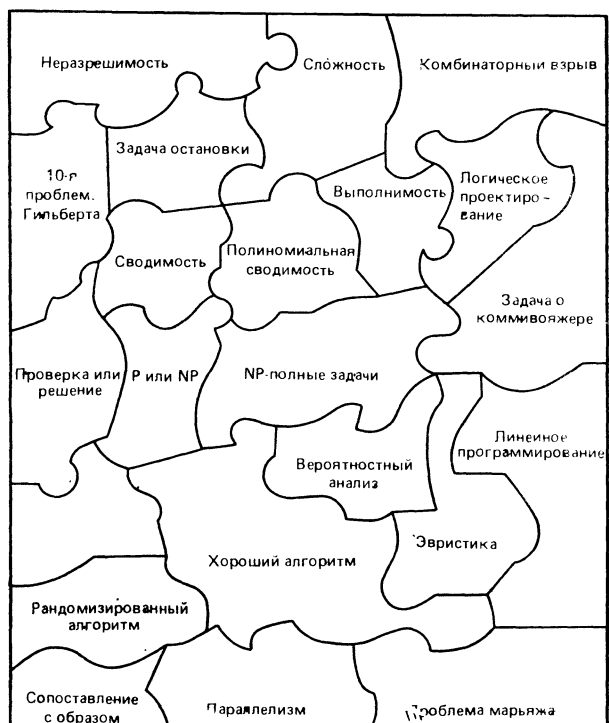
Временами два уровня соприкасаются. Такие соприкосновения обычно очень важны. В науке взаимодействие двух областей, близость которых до этого не замечали, приводит к появлению большого числа важных работ. Абстрактное изучение классов сложности связывается со свойствами конкретных

задач — вроде задачи о коммивояжере или проблемы выполнимости — посредством понятия  $NP$ -полноты.

**К.Ф.** Сделанный вами шаг в сторону вероятностного анализа явился отходом от парадигмы анализа для худшего случая. И вы продолжали разрабатывать вероятностный подход вопреки его хулителям. Чем было мотивировано ваше решение?

**Р.К.** Я не хочу создать впечатление, что до меня о вероятностном подходе никто ничего не слышал. Он, бесспорно, применялся, но в основном к задачам сортировки, поиска и структур данных, а не к задачам комбинаторной оптимизации.

Мое решение было особенно трудным потому, что до некоторой степени я был согласен с хулителями. Налицо действительно фундаментальная методологическая проблема: как выбирать распределения вероятностей? Откуда вы можете знать, какова будет статистика частных случаев задачи? Никто никогда не измерял тщательно характеристики задач из реального мира, и даже если бы измерили, то все равно лишь для одной среды вычислений. Но выхода я не видел, ведь не пойдешь **мы** по веро-



ятностной дороге,  $NP$ -полнота оказалась бы просто опустошительной.

Кроме того, было еще одно направление — исследование приближенных алгоритмов, не гарантирующих результата. Имея перед собой  $NP$ -полную задачу оптимизации, можно ослабить требование получения оптимального решения и попытаться построить быстрый алгоритм, гарантирующий для худшего случая результат, который хуже оптимального, скажем, не более чем на 20 процентов. Это еще одна очень интересная парадигма, она исследовалась, и результаты были неоднородны. Для некоторых задач трудности действительно были устранены — можно получить решение, сколь угодно близкое к оптимуму. Для некоторых других можно гарантировать попадание с точностью 22, 23 или 50 процентов. Результаты эти были очень изящны, но, по-моему, не характеризовали то, что происходит при применении практических эвристик. Практические эвристики очень хороши почти всегда, но не в худшем случае.

Так что не очень-то я рвался разрабатывать новое направление, основания которого можно было бы оспорить. Это был и личный риск, поскольку меня могли счесть слабаком. Ну, знаете: «Задачу решить не может, так вводит вероятностные предположения и облегчает себе жизнь». Но иного пути я опять-таки не видел. Некуда было деваться от явления  $NP$ -полноты.

**К.Ф.** Не имея оптимального решения задачи, откуда вы знаете, что ваша эвристика дает решение, близкое к оптимальному?

**Р.К.** Это действительно методологическая трудность. Когда выполняешь эвристический алгоритм и он вроде бы дает очень хорошие решения, уверенности все равно быть не может, поскольку не знаешь, где лежит оптимум. Можно запускать программу из различных начальных точек и получать все время одно и то же решение. Если никто никак не найдет лучшего решения — это косвенное доказательство, что ваше решение наилучшее. Можно также затратить очень много машинного времени на вычисление методом ветвей и границ и получить в конце концов решение, про которое сумеете доказать, что оно хуже оптимального не более чем на полпроцента. Затем вы запускаете на три минуты для той же задачи вашу быструю эвристику и смотрите, насколько она может приблизиться к первому решению. Иногда можно построить задачу искусственно, так что будешь знать оптимальное решение. Но вы указали на действительно серьезную методологическую проблему.

**К.Ф.** Недавно вы начали работу над параллельными вычислениями. Каким образом вопросы параллельной обработки влияют на наше понимание того, что такое хороший или эффективный алгоритм?

**Р.К.** Параллельные вычисления меня чрезвычайно интересуют, и, по-моему, эта область очаровательная. Есть несколько направлений исследований, между которыми еще не вполне установились связи; изучаются разные типы параллельной архитектуры, существует множество вопросов о том, какими должны быть процессоры и как они должны быть взаимосвязаны. Есть также вопросы численного анализа, вопросы сложности и синтеза алгоритмов.

За последние два года значительная часть моей работы выполнена в сотрудничестве с двумя израильскими коллегами, Ави Вигдерсоном и Эли Упфалом. Мы изучали сложность параллельных алгоритмов в плане довольно-таки теоретическом, работая с идеализированными моделями параллельных компьютеров. Этим мы абстрагируемся от определенных вопросов связи между компонентами, возникающими из-за того, что параллельная система — это на самом деле также и распределенная система. Например, мы можем предполагать, что возможно прямое общение между любыми двумя процессорами, что просто неверно. Но это полезные абстракции, позволяющие нам работать над некоторыми структурными вопросами, например: «Что именно в задаче поддается параллельной обработке? При каких обстоятельствах мы можем построить совершенно новый алгоритм, который в огромной степени уменьшит время, необходимое для решения задач?»

Это интересные и важные упражнения для ума, потому что они подводят к открытию совершенно новой методики структурирования алгоритмов. Очень часто предлагаемые нами параллельные алгоритмы совершенно непохожи на последовательные алгоритмы для тех же задач.

Работая над параллельными вычислениями, специалисты по информатике в основном заняты дальнейшим убыстрением алгоритмов, и так полиномиальных по времени. Мы же задаемся другими вопросами: «Какие задачи этого класса можно распараллелить в колоссальной степени? При каких условиях возможно огромное сжатие вычислений?» В будущей работе я буду больше внимания уделять применению распараллеливания к  $NP$ -полным задачам. Человек, настроенный строго теоретически, возможно, скажет: «Безнадежное дело, экспоненциальное время не уменьшить до полиномиального за счет привлечения многих процессоров, разве что у вас экспоненциально растущее число процессоров». С другой стороны, пусть даже никогда не удастся перейти от экспоненты к полиному, ясно и то, что для этих задач есть огромные возможности для распараллеливания, и оно может все-таки помочь нам справляться с комбинаторными взрывами.

Я намереваюсь проанализировать метод ветвей и границ,

деревья игр, структуры «цель — подцель», прологоподобные структуры, поиск с возвратами и все разнообразные виды комбинаторного поиска, потому что такие задачи, по-моему, очень хорошо подходят для параллельных вычислений. Пока не ясно, какую именно форму примет эта теория.

**К.Ф.** Какова связь между вашим теперешним интересом к параллелизму и прежней работой с Раймондом Э. Миллером?

**Р.К.** Было два основных периода, когда я занимался изучением параллельных вычислений. Первый — в первой половине шестидесятых годов, когда я работал с Миллером над рядом формализмов для описания параллельных вычислений. Второй — сравнительно недавний, когда я работал с Упфалом и Вигдерсоном над построением и анализом параллельных алгоритмов. Исходным вопросом для нас с Миллером был вопрос о том, нужно ли и можно ли сконструировать аппаратные средства специального назначения для параллельного выполнения часто применяющихся итеративных вычислений. Мы предложили несколько формализмов для описания параллельных вычислений. Один был вполне конкретным, а еще один очень теоретический по своему характеру. Модели и методы, предложенные нами, по духу очень походили на «систолические схемы», выдвинутые впоследствии Х. Т. Кунгом, Чарлзом Лейзерсоном и другими, хотя я не хочу сказать, что мы превзошли все их идеи. Многого мы, конечно, не увидели, но в некотором смысле мы делали это слишком рано — мир не вполне был к этому готов.

Нас интересовали и некоторые вопросы более качественного характера, например: «Что происходит, если работа идет асинхронно и нет общих часов, так что нельзя сказать, происходит ли А раньше В или В раньше А? Можно ли все же получить вполне определенный результат, даже если невозможно управлять порядком, в котором происходят различные события в параллельных процессах?»

Работа последнего времени идет в другом направлении. Нас интересует сложность: насколько быстро можно решить задачу при заданном числе процессоров? Эти два направления очень разные.

**К.Ф.** Как по-вашему, сможет ли ваша работа способствовать также конструированию параллельных процессоров и помочь определению наилучших способов связывания разных процессоров внутри машины?

**Р.К.** На схемном уровне то, что я делаю, имеет непосредственное отношение к таким работам. Разработка чипа для интегральной схемы немного похожа на проектирование города на 50 000 жителей. Налицо всевозможные комбинаторные проблемы, связанные с размещением разных модулей схемы и трас-

сировкой соединений между ними. На уровне архитектурном моя работа по параллельным вычислениям непосредственно неприменима, потому что я пользовался идеализированными моделями, в которых игнорировались проблемы связи между процессорами. Надеюсь, что моя работа станет ближе к вопросам архитектуры.

**К.Ф.** Можно ли мы из теоретических исследований узнать что-нибудь воодушевляющее о распределенной связи и распределенных протоколах?

**Р.К.** Да. Есть очень красивые теоретические разработки о том, сколько теряешь, когда вынужден полагаться на передачу сообщений в разреженной сети процессоров вместо прямой попарной связи между процессорами. В реалистичной распределенной системе процессоры должны не только считать, но и сотрудничать подобно почте, где сообщения летают между местами обработки. Есть очень хорошие теоретические работы о различных типах протоколов, где, как в так называемой задаче о византийских генералах, — еще одно из этих громких названий, — коллектив процессоров должен прийти к соглашению посредством передачи сообщений, даже когда часть процессоров неисправна и действует враждебно, пытаясь все запутать. Уже стало ясным, что здесь очень хорошо работает рандомизация. Протоколы, нужные для этих задач сотрудничества и связи в распределенной системе, можно упростить, если использовать бросание монеты. Что в такого рода задачах применимы рандомизированные алгоритмы — это важное открытие. Так что есть много связей между теорией и протоколами для реальных распределенных систем.

**К.Ф.** Применяются и такие алгоритмы, которые, хотя и недостаточно обоснованы теоретически, на практике работают очень хорошо. И практики могли бы сказать: «Эти результаты очаровательны, но раз мы можем посредством проб и ошибок наткнуться на прекрасно работающие алгоритмы, зачем же заботиться о теории?» Как ваша работа будет применяться в будущем в самом практическом смысле?

**Р.К.** Некоторые из важнейших комбинаторных алгоритмов никогда не удалось бы изобрести в процессе проб и ошибок, правильный теоретический подход был абсолютно необходим. Когда общая форма алгоритма определилась, часто бывает возможно настроить его эмпирически, но если действовать чисто эмпирически, то ваши знания ограничены теми весьма конкретными обстоятельствами, в которых вы экспериментируете. Результаты же анализа в большей степени поддаются обобщению. Оправданием теории, помимо ее очевидной эстетической привлекательности, служит то, что, когда вы получаете теоретический результат, он обычно приложим к целому классу



ситуаций. Это напоминает соотношение моделирования и анализа. Конечно, свое место есть и у того и у другого, но результаты моделирования по большей части говорят вам об одной очень ограниченной ситуации, в то время как анализ иногда может сказать вам о целом классе ситуаций. Но решение задач комбинаторной оптимизации, бесспорно, в такой же степени искусство, как и наука, и есть люди с чудесно отточенной интуицией для построения эвристических алгоритмов решения задач.

**К.Ф. На чем будут сосредоточены исследования в Институте математических наук (ИМН)?**

**Р.К.** Рад, что вы меня спросили об ИМН, потому что этот проект мне очень близок. Это исследовательский институт в горах за университетским городком Беркли, но официально он не связан с университетом, и там финансируются годовые исследовательские программы в математических науках. В прошлом эти программы в основном были по чистой математике. Средства выделяет в первую очередь Национальный научный фонд (ННФ).

Около двух лет тому назад мы со Стивеном Смейлом из математического факультета в Беркли предложили годичный проект по вычислительной сложности, и мы были очень довольны, что его приняли. По-моему, это указывает на то, что математики, которые не торопились заниматься вопросами вычислительной сложности, сейчас стали к ней очень восприимчивы. В этой исследовательской работе по теории сложности будут участвовать около 70 специалистов. Это число поровну поделено между математиками и специалистами по информатике.

Я очень горжусь группой, которую мы собрали. Люди занимаются широким спектром задач. Некоторые — метатеорией, сосредотачиваясь не на конкретных задачах, а на классах вроде  $P$  и  $NP$ . Некоторые — вычислительной теории чисел, где центральная проблема — разложение очень больших чисел на множители. Другие сосредоточились на комбинаторных задачах. Мы исследуем пограничные вопросы между численным анализом и теорией сложности. И важная тема — параллельные вычисления. Я просто в восторге от того, как идут дела — там прямо-таки Мекка для специалистов по теории сложности. Всего за пару месяцев, пока мы работаем, уже кое-что получено для параллельных алгоритмов.

**К.Ф. Нельзя ли поконкретнее?**

**Р.К.** Приводить конкретные результаты было бы преждевременно; скажу лишь, что некоторые из них заставляют воспринимать мои ранние работы как устаревшие.

**К.Ф. Сколько денег выделено на этот проект ИМН?**

**Р.К.** Бюджет проекта по сложности в ИМН составляет ок-

ругленно 500 тыс. долл. от Национального научного фонда и 140 тыс. долл. от военных ведомств.

Этот проект для теории сложности оказался своего рода неожиданным подарком судьбы, но я очень отчетливо чувствую: что в целом информатика давно не финансировалась так плохо. Национальный научный фонд поддерживает несколько очень достойных новых инициатив, но это делается без соответствующего расширения финансовой основы, так что эти инициативы субсидируются за счет уже существующих программ. Хотя я должен сказать, что программа ИМН является исключением, в целом люди, занимающиеся теоретической информатикой, потеснены уменьшением финансирования вследствие смещения приоритетов ННФ — в основном в сторону инженерных исследований.

**К.Ф.** В чем заключается практический интерес Министерства обороны и трех этих ведомств?

**Р.К.** Поддержка, которая исходит от них, касается в основном параллельных и распределенных вычислений, и мы планируем провести весной рабочее совещание, которое соберет вместе математиков, специалистов по численному анализу и компьютерной архитектуре. Конечно, в наши дни бывают всевозможные мероприятия по суперкомпьютерам и параллельным вычислениям, а это должно способствовать более тесному взаимодействию между теорией сложности и более реальными проблемами пользователей и конструкторов.

**К.Ф.** Было много дебатов о достоинствах Стратегической оборонной инициативы (СОИ). Не хотели бы вы их прокомментировать?

**Р.К.** Я не намерен ни выступать на тему о звездных войнах, ни претендовать на то, чтобы быть экспертом по построению программ. Но я изучил некоторые данные, свидетельствующие о том, что опасно строить распределенную систему беспрецедентных масштабов, которая не может работать или быть проверенной до критического момента. Я убежден этими аргументами, и лично я в этом участвовать не буду.

**К.Ф.** Изучением сложности заняты исследователи во многих областях. Не могли бы вы проиллюстрировать, если возможно, отношения между изучением сложности в информатике и в других дисциплинах?

**Р.К.** Сложность означает много разных вещей — существует дескриптивная сложность и вычислительная сложность. Алгоритм может быть чрезвычайно сложным в смысле способа его построения и при этом работать очень быстро, так что его вычислительная сложность низка. Таким образом, мы имеем различные понятия о сложности. Мне неясно, имеют ли в виду одно и то же понятие инженеры-электронщики, экономисты,

математики, специалисты по информатике и физики, употребляя термин сложность. Однако я думаю, что имеется несколько стоящих и интересных аналогий между вопросами сложности в информатике и экономике. Например, в экономике традиционно предполагается, что субъекты экономики имеют универсальную вычислительную мощность и мгновенно узнают, что происходит в остальной экономике. Специалисты по информатике отрицают, что алгоритм может иметь бесконечную вычислительную мощность. Они фактически изучают ограничения, возникающие из-за вычислительной сложности. Таким образом, здесь явная связь с экономикой.

Кроме того, можно сказать, что традиционная экономика — здесь я в самом деле выхожу за рамки своей специальности — не учитывает задержку информации, а также то, что мы принимаем решения, не располагая полной информацией об экономических возможностях, открытых для нас. Это очень похоже на то, что имеет место в сетях связи: узел в распределенной компьютерной сети видит только свое непосредственное окружение и посылаемые ему сообщения. Таким образом, аналогия неоспорима, но надо быть осторожным, потому что мы не всегда имеем в виду одно и то же, когда рассуждаем о сложности.

**К.Ф.** Используете ли вы понятие «эвристика» иначе, чем делают это исследователи ИИ?

**Р.К.** Специалисты по ИИ различают алгоритмы и эвристики. Я думаю, что это все — алгоритмы. Для меня алгоритм — это просто любая процедура, которая может быть выражена на языке программирования. Эвристики — это всего лишь алгоритмы, которые мы понимаем не слишком хорошо. Я стремлюсь обитать в искусственно точном мире, где я точно знаю, что мой алгоритм должен делать. Ну а когда вы говорите о программе, которая должна хорошо играть в шахматы, переводить с русского на английский или решать, что заказать в ресторане, — я упомянул несколько задач с привкусом ИИ, — ясно, что требования к ней гораздо расплывчатее. Это характерно для тех программ, которые в ИИ считаются эвристическими.

**К.Ф.** Дэвид Парнас указывает в последней статье, что системы, разработанные в рамках эвристического программирования, т.е. программирования путем проб и ошибок в отсутствие точной спецификации, в принципе менее надежны, чем программы, созданные более формальными методами.

**Р.К.** Да, и это возвращает нас к СОИ. Это одна из причин относиться к ней с тревогой. Я думаю, что мы располагаем гораздо более сильным аппаратом отладки программ, когда мы по крайней мере определили, что программа должна делать.

**К.Ф.** Некоторые члены сообщества исследователей ИИ отве-

чают на эту критику утверждением, что они пытаются моделировать человека и что люди не имеют четких спецификаций. Самое лучшее, на что они могут надеяться, — это моделирование ненадежной системы.

**Р.К.** Я в самом деле верю в попытки, основанные на жестких гипотезах и жестких заключениях. Я понимаю, что в некоторых областях компьютерной науки должны преобладать эмпирические исследования, но это не освобождает нас от ответственности серьезно думать о том, что мы измеряем, чего мы пытаемся достичь и когда мы можем сказать, что наша конструкция оказалась успешной. И я думаю, что в некоторой мере необходим научный метод. Для меня неприемлема идея, что просто потому, что вы моделируете непознаваемый до конца процесс человеческого познания, вы освобождены от обязательств точно формулировать, что вы делаете.

**К.Ф.** Что, по вашему мнению, вообще представляет собой информатика как дисциплина?

**Р.К.** Информатика имеет огромные преимущества по причине колоссальной важности и привлекательности этой области сегодня. По определенным меркам мы достигли немалых успехов. Значительная часть талантливой молодежи страны тянется в нашу область, особенно в направления искусственного интеллекта и теоретической информатики.

Если же говорить о развитии нашей области как науки, то мне кажется, что мы являемся в некоторой мере жертвами собственных успехов. Существует столько способов заработать деньги, так много вещей, которые хочется попробовать, так много увлекательных направлений, что мы иногда забываем подумать об основаниях нашей дисциплины. Нам нужно постоянно поддерживать равновесие между состоянием, когда мы даем волю своим порывам к изготовлению всевозможных искусных поделок, и планированием наших экспериментов научными методами, гарантирующими нормальное развитие фундаментальных исследований. Инструменты наши столь мощны, перспективы столь велики, диапазон возможных приложений столь огромен, что есть большой соблазн пахать все дальше. И пахать надо. Но надо также помнить, что мы — научная дисциплина, а не просто ветвь высокой технологии.

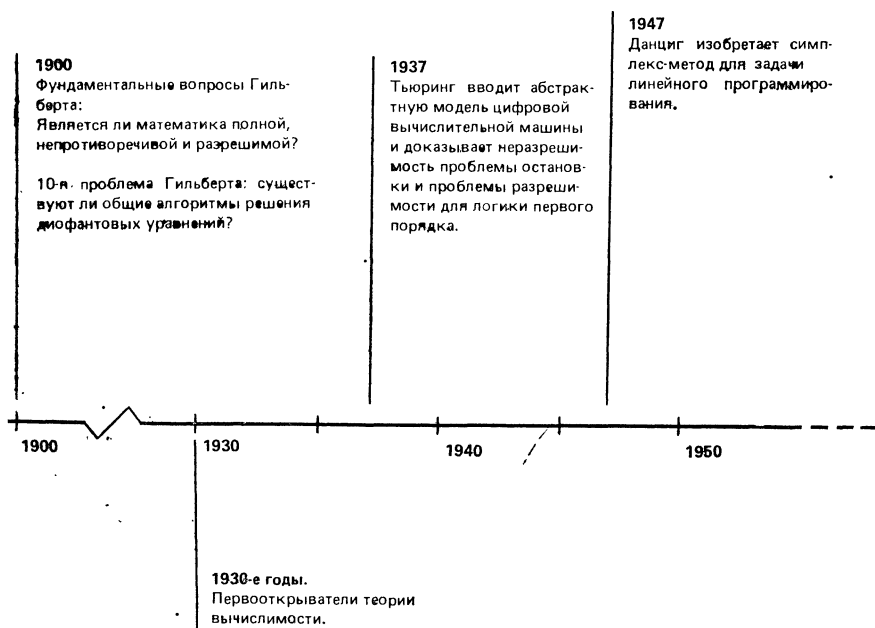
**К.Ф.** Вы обратили внимание на важность сочетания искусства и науки, проникательности и интуиции, равно как и более строгих методов исследования. Бывало ли, что что-то вам просто являлось и вы испытывали так называемый феномен «эврика!», о котором говорят изобретатели?

**Р.К.** Я думаю, что мы все это испытывали — утром просыпаешься с готовым решением задачи. Надо помнить, что этим «эврикам» обычно предшествует большой тяжелый труд, кото-

рый иногда кажется непродуктивным. Например, когда я прочел статью Кука 1971 г., я довольно скоро понял, что он нашел ключ к вещам феноменально важным, и начал работать дальше, пытаясь продемонстрировать размах и значение его результатов. В некотором смысле это было почти мгновенно, но было подготовлено более чем десятилетием работы. По-моему, характерно, что такие моменты, когда устанавливаются связи, наступают после долгого подготовительного периода.

**К.Ф.** Бывало ли у вас, что вы с кем-нибудь беседуете и от случайного замечания у вас в голове — «щелк!»?

**Р.К.** О да. Мне бывает очень полезно объяснить, чем я занимаюсь, потому что мои ошибки становятся мне ясны гораздо быстрее. А других я слушаю, потому что в самом деле верю в



пользу расширения своей базы знаний. Это сильно увеличивает вероятность обнаружить впоследствии неожиданные связи.

## РАЗВИТИЕ КОМБИНАТОРНОЙ ОПТИМИЗАЦИИ И ТЕОРИИ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ

**1965**

Харманис и Стирнз определяют понятие "сложность", вводя подход к вычислительной сложности посредством абстрактных машин, и получают результаты о структуре классов сложности.

Эдмондс определяет "хороший" алгоритм как такой, время работы которого ограничено полиномиальной функцией от размера входа. Находит такой алгоритм для проблемы марьяжа.

**1957**

Форд и Фалкерсон с сотр. предложили эффективный алгоритм для проблемы потоков в сетях.

**1960**

**1959**

Рабин, Макнотон и Ямада: первые проблески теории вычислительной сложности

**1970-е годы**

Поиск почти оптимальных решений, основанный на верхней оценке отношения стоимости данного решения к стоимости оптимального решения.

**1971**

На основе работы Дэвиса, Робинсона и Путмана Матиясевич решает 10-ю проблеме Гильберта: не существует общей процедуры решения диофантовых уравнений.

Теорема Кука: все NP-полные проблемы полиномиально сводимы к проблеме выполнимости

Левин также открывает этот принцип.

**1970**

**≈ 1980**

Боргвардт, Смейл и др. выполняют вероятностный анализ симплекс-метода

**1984**

Кармаркар изобретает теоретически эффективный и практически удобный алгоритм линейного программирования.

**1980**

**1976**

Рабин и др. приступают к изучению рандомизированных алгоритмов

**1975**

Карп отходит от парадигмы изучения худшего случая и разрабатывает вероятностный анализ комбинаторных алгоритмов.

**1973**

Мейер, Стокмейер и др. доказывают практическую неразрешимость некоторых проблем разрешения в логике и теории автоматов.

**1972**

Карп использует полиномиальную сводимость, чтобы показать, что 21 проблема (упаковки, паросочетания, замощения и др.) NP-полные.

# Биографии авторов

## Charles W. Bachman

Чарльз Бахман получил Тьюринговскую премию в 1973 г. за работы в области методов создания баз данных, выполненные для компаний «Дженерал Электрик» и «Хонейуэлл». Ему принадлежат многочисленные патенты, относящиеся к системам управления базами данных и моделям данных. Им разработана система Integrated Data Store (IDS), послужившая основой систем управления базами данных семейства CODASYL (IDMS, DMS 1100 и многих других). Он играл ведущую роль в разработке ISO Reference Model for Open Systems Interconnections, будучи председателем комитета ISO/TC97/SC16. Ныне он является президентом компании «Бахман информейшн системс инкорпорейтед», которая была основана в 1983 г. для обеспечения разработчиков программного обеспечения программными продуктами, поддерживающими весь цикл создания и сопровождения прикладных пакетов программ с помощью методов баз данных, системного программирования и искусственного интеллекта.

## John Backus

Джон Бэкус был награжден Тьюринговской премией в 1977 г. за разработку языка программирования Фортран и языка описания синтаксиса БНФ (форма Бэкуса—Наура). Приняв участие во многих различных проектах в Исследовательской лаборатории фирмы IBM в Сан-Хосе, Калифорния, он разработал функциональный стиль программирования, основанный на использовании комбинационных форм для создания программ.

Ныне его научные интересы включают типы данных для функционального программирования, алгебраические преобразования программ и оптимизацию. Совместно с Джоном Уильямсом и Эдвардом Уимерсом он разработал новый язык программирования FL. Это универсальный язык, поддерживающий общие файлы ввода — вывода и интерактивные программы. В этом языке особое внимание уделено точному семантическому описанию и обработке абстрактных типов данных. В настоящее время вместе со своими сотрудниками он разрабатывает оптимизирующий компилятор, использующий алгебраические преобразования.

## E. F. Codd

Эдгар Кодд был членом ученого совета Исследовательской лаборатории фирмы IBM в Сан-Хосе, Калифорния, когда ему была присуждена Тьюринговская премия 1981 г. за его вклад в теорию и практику создания систем управления базами данных. Кодд заинтересовался управлением большими коммерческими базами данных в 1968 г. и начал с создания реляционной модели в качестве основы таких баз данных. В начале 70-х годов он обогатил эту работу, разлив методы нормализации для проектирования баз данных, а также разработав два совершенно различных языка — один алгебраический, а другой — основанный на логике предикатов.

В 1985 г. Кодд вместе со своим коллегой Дейтом основал в Сан-Хосе

две новые компании: Реляционный институт (The Relational Institute), организующий и проводящий профессиональные семинары по реляционному подходу к управлению базами данных, и Консультационную группу Кодда и Дейта (Codd and Date Consulting Group), предлагающую консультации по вопросам управления базами данных.

### **Stephen A. Cook**

Стефен Кук, лауреат премии Тьюринга 1982 года, получил ученую степень доктора философии в области математики в Гарвардском университете в 1966 г. и вскоре стал сотрудником факультета математики университета штата Калифорния в Беркли. В 1970 г. он переходит на факультет информатики университета г. Торонто, где читает курсы информатики студентам младших и старших курсов. Он также проводит научные исследования по вычислительной сложности и теории конструктивно осуществимых доказательств, которые он описывает как важнейшие способы связать воедино математическую логику и теорию сложности.

Все его исследования мотивированы стремлением найти нижние границы сложности вычислительных проблем. Его работы по NP-полноте принесли ему особенное удовлетворение.

### **Edsger W. Dijkstra**

Эдсгер Дейкстра работал в Эндховенском технологическом университете в Голландии, где он преподавал и разрабатывал императивный язык программирования, на котором он научился писать прекрасные программы, когда в 1972 г. он был удостоен Тьюринговской премии. В 1973 г. он становится членом исследовательской группы «Борроуз рисерч» (Burroughs Research) и в течение последующего десятилетия пишет более 500 технических отчетов по различным исследовательским проектам. В 1984 г. он принимает приглашение стать профессором и заведовать кафедрой информатики им. Шлumberгера в Университете штата Техас в Остине. В настоящее время он работает, в частности, над выводом и представлением программ и оптимизацией математических рассуждений.

Во всех своих исследованиях Э. Дейкстра придает большое значение простоте и изяществу математических рассуждений. При написании своих работ он создал новый стиль научных и технических сообщений, который можно описать как нечто среднее между журнальными публикациями и дружеской перепиской.

### **Robert W. Floyd**

Роберт Флойд, награжденный Тьюринговской премией в 1978 г., является профессором информатики в Станфордском университете с 1968 г. Сейчас его научные интересы включают исправление синтаксических ошибок и анализ алгоритмов. Он надеется когда-нибудь завершить учебник, служащий введением в теорию алгоритмов, и еще один, посвященный тому, что компьютеры (реальные или воображаемые) могут и чего они не могут делать. Он убежден, что математика является самым лучшим инструментом серьезного пользователя компьютера.

### **R. W. Hamming**

Ричард Хэмминг получил Тьюринговскую премию 1968 г. за работу по кодам, исправляющим ошибки, которую он проделал для AT&T Bell Laboratories. В течение 30 лет он работал в этой компании над множеством исследовательских проектов. В 1976 г. он перешел в Naval Postgraduate School, Монтерей, Калифорния, в качестве профессора-адъюнкта. Там он преподает и пишет книги по теории вероятностей и комбинаторике.



Хэмминг — сторонник разностороннего научного образования, который стремится учить своих студентов достигать в науке совершенства, а не просто овладевать техническими навыками.

### **Charles Antony Richard Hoare**

Чарльз Энтони Ричард Хоар начал свою карьеру в 1960 г. в Англии, где он писал программы для компании «Эллиот Бразерс Лтд». За восемь лет, проведенных им в этой фирме, он занимался разработкой всевозможных программных продуктов, от простых подпрограмм до языков программирования высокого уровня. Он покинул деловой мир в 1968 г., чтобы преподавать информатику в Королевском Университете в Белфасте, Ирландия. В 1977 г. он перешел на факультет информатики Оксфордского университета, где занимает должность профессора вычислительной математики.

В 1980 г. он был награжден премией Тьюринга за его вклад в формальное определение языков программирования посредством аксиоматической семантики. Превращение программирования в серьезную профессиональную дисциплину стало ведущим мотивом его научной деятельности.

Хоар указывает на разработку концепции взаимодействующих последовательных процессов как на главную область своих научных интересов. В настоящее время его интересуют перенос методов, базы данных и математические методы информатики.

### **Kennet E. Iverson**

Кеннет Айверсон, будучи аспирантом, а затем ассистентом Гарвардского университета, предложил аналитический язык программирования APL в качестве основы лучшего математического языка, а также как попытку обеспечить более ясную и точную форму записи выражений при обучении и написании математических работ. Когда в 1960 г. он начал работать в Исследовательском центре им. Томаса Дж. Уатсона компании IBM, он убедил нескольких своих коллег присоединиться к нему в дальнейшей разработке и реализации языка APL. В 1980 г. он перешел из IBM в компанию I. P. Sharp Associates в Торонто, которая занимается предоставлением услуг и продуктов, связанных с языком APL, в основном среди банковских кругов. Недавно он ушел из этой компании, чтобы заняться применением APL в системе образования, в качестве средства обучения программированию и другим связанными с этими начинаниями.

### **Richard M. Karp**

Ричард Карп защитил свою докторскую диссертацию по прикладной математике в Гарвардском университете в 1959 г. Затем он занимается научной работой в Исследовательском центре им. Томаса Дж. Уотсона компании IBM вплоть до 1968 г., совмещая ее с должностью приглашенного профессора в Нью-Йоркском университете, университете штата Минчиган и в Политехническом институте Бруклина. Ныне он профессор информатики, исследования операций и математики в Калифорнийском университете в Беркли.

Карп — лауреат Тьюринговской премии 1985 г., ведущий специалист в теории сложности вычислений. Его последние работы по комбинаторным алгоритмам и NP-полноте изменили подход специалистов по информатике к таким практическим задачам, как трассировка, упаковка, покрытия, сопоставление с образцом и задача о коммивояжере. В настоящее время он пишет книгу о вероятностном анализе комбинаторных алгоритмов.

### **Donald E. Knuth**

Дональд Кнут как раз закончил третий том своего семитомника «Искусство программирования для ЭВМ», когда в 1974 г. ему была присуждена Тьюринговская премия. Его личный опыт написания книг и неудовлетворен-

ность тем, в каком виде были представлены гранки, возбудили в нем интерес к компьютерному набору. Этот интерес привел к созданию алгоритма компьютерного набора TeX и алгоритма проектирования шрифтов «Метафонт».

В настоящее время Кнут является профессором информатики Станфордского университета и работает над четвертым томом своего семитомника. Назначение этих книг — научить читателя составлять лучшие алгоритмы, вместо того чтобы лучше использовать чужие. Кнут убежден, что программирование для компьютера может приносить эстетическое удовлетворение, подобно сочинению музыки или стихов.

### **John McCarthy**

Джон Маккарти заинтересовался проблемами искусственного интеллекта в 1949 г., будучи аспирантом-математиком. Ныне он профессор информатики на факультете информатики, а также профессор кафедры им. Шарля М. Пижо Инженерной школы Станфордского университета. Его главный научный интерес — формализация так называемых «знаний здравого смысла». Он изобрел Лисп в 1958 г., разработал концепцию разделения времени и начиная с первой половины 60-х годов работает над доказательством того, что компьютерные программы соответствуют техническим заданиям на их разработку. Его последним теоретическим достижением является метод ограничений для немонотонных рассуждений. Он был награжден Тьюринговской премией в 1971 г. за достижения в области искусственного интеллекта.

### **Marvin Minsky**

Марвин Минский работал над устройством Robot C — первым роботом, руки которого должны были двигаться подобно человеческим, когда он был награжден премией Тьюринга в 1969 г. Он продолжает преподавать и проводить исследования в Массачусетском технологическом институте (MIT), где он является профессором факультета электроники и информатики. Его научная деятельность связана с рядом проектов, в основном в области искусственного интеллекта, включая математическую теорию вычислений и робототехнику. Он основал лабораторию искусственного интеллекта в MIT, а также Logo Computer Systems, Inc., и Thinking Machines, Inc.

Он также выступал в качестве консультанта исследовательских проектов столь несхожих организаций, как НАСА и Национальный институт танца. Его научные интересы включили распознавание музыки и физическую оптику. Он разработал объяснение способа, посредством которого должна действовать «думающая машина», и изложил свои идеи в недавно опубликованной книге «The Society of Mind» (издательство «Саймон и Шустер»).

### **Allen Newell**

Аллен Ньюэлл, награжденный Тьюринговской премией 1975 г. вместе с Хербертом Саймоном, начинал свою научную деятельность в 50-х годах в «Рэнд Корпорейшн». В 1961 г. он перешел в Университет Карнеги — Меллона, где он является профессором информатики. Его исследования сконцентрированы на решении задач и методологии познания применительно к искусственному интеллекту и когнитивной психологии.

Достижения Ньюэлла в информатике включают обработку списков, дескриптивные языки программирования и основанные на результатах психологических исследований модели взаимодействия человека и компьютера. В настоящее время его деятельность и научные интересы включают разработку архитектур для решения задач и обучения, единую теорию интеллекта и архитектуру компьютеров для систем ИИ, основанных на правилах логического вывода.

**Alan J. Perlis**

Алан Перлис был первым лауреатом Тьюринговской премии, которая была присуждена в 1966 г. В 1952 г. он основал Центр цифровых компьютеров в Университете Пердью, а в 1956 г. — Вычислительный центр в Технологическом институте им. Карнеги (ныне Университет Карнеги—Меллона). В 1965 г. он основал первое отделение аспирантуры по информатике в Технологическом институте им. Карнеги и стал его первым заведующим. Перлис участвовал в стандартизации языков Алгол-58 и Алгол-60, а также в последующих расширениях этого языка. В 1971 г. он стал профессором информатики факультета информатики Йельского университета. Большая часть его исследований связана с разработкой языков программирования и методов составления программ.

Перлис является основателем и редактором журнала «Communications of the ACM»; он был президентом ACM с 1962 по 1964 г. В настоящее время его интересуют языки программирования для параллельных вычислений и динамическое поведение программных систем.

**Michael O. Rabin**

Микаэль Рабин в настоящее время занимает две должности: он профессор информатики в Гарвардском университете и профессор информатики и математики в Еврейском университете в Иерусалиме. Основным мотивом его научной работы было изучение теории алгоритмов с особым вниманием к прямым приложениям в компьютерной технологии (некоторые рандомизированные алгоритмы, над которыми он работал, нашли применение в защите компьютеров от несанкционированного доступа и в защищенных операционных системах). Солауреат Тьюринговской премии 1976 г., он был основоположником теории сложности вычислений и понятия недетерминированных вычислений.

Рабин ввел вероятность в теорию вычислений, начиная со статьи о вероятностных автоматах. Затем он работал над проверкой больших чисел на простоту и многими другими приложениями методов рандомизации к теории вычислений. Его работа по древесным автоматам поставила много открытых проблем разрешимости в математической логике.

**Dennis M. Ritchie**

Дэнис Ритчи, награжденный премией Тьюринга 1983 г. вместе с Кеном Томпсоном, стал штатным сотрудником AT&T Bell Laboratories, Мюррей Хилл, Нью-Джерси, в 1968 г., где он продолжает разрабатывать языки программирования и операционные системы. Группа Ритчи и Томпсона больше всего известна как создатели и реализаторы операционной системы UNIX и разработчики языка программирования Си, на котором написана эта система. Ритчи также участвовал в разработке системы MULTICS.

**Dana S. Scott**

Дана Скотт является профессором информатики, математической логики и философии университета Карнеги—Меллона, занимая эту должность с 1981 г. Он был награжден Тьюринговской премией (совместно с Микаэлем Рабином) в 1976 г., будучи профессором математической логики в Оксфордском университете, Оксфорд, Англия.

Работы Скотта в области логики касались теории моделей, теории автоматов, теории множеств, модальной и интуиционистской логики, конструктивной математики и связей между логикой и теорией категорий. В настоящее время он интересуется широким кругом вопросов, связанных с приложениями логики к семантике языков программирования и с вычислительной лингвистикой.

**Herbert A. Simon**

Херберт Саймон является профессором информатики и психологии университета Карнеги — Меллона. Его исследования по теории интеллекта начались в 1949 г. и включали использование компьютеров для моделирования человеческого мышления и решения задач. Он был награжден Тьюринговской премией 1975 г. совместно с Алленом Ньюэллом за работу, показывающую, как использовать эвристический поиск для решения задач.

Начиная с середины 70-х годов Саймон проводил исследования в трех основных областях: психологии научных открытий, что включало написание компьютерных программ, имитирующих процесс открытия; методов обучения, причем для моделирования того, как студенты учатся, использовались системы продукций; изучении того, как люди представляют и понимают то, что им исходно объясняют на естественном языке. В последнее время он изучает, как люди воспринимают словесные утверждения и преобразуют их в визуальные образы. Почему для людей это важно, т. е. почему лучше один раз увидеть, чем сто раз услышать, является темой и названием его недавней статьи в журнале «Cognitive Science».

**Ken Thompson**

Кен Томпсон, штатный сотрудник AT&T Bell Laboratories, Мюррей Хилл, Нью-Джерси, получил Тьюринговскую премию 1983 г. вместе с его коллегой Деннисом Ритчи. В центре его исследовательских интересов находились компиляторы, языки программирования и операционные системы. Вместе с Ритчи он стоял у истоков хорошо известной операционной системы UNIX.

Томпсон написал множество опубликованных статей и отчетов по широкому кругу вопросов — от разработки операционных систем и создания алгоритмов до компьютерных программ, умеющих играть в шахматы. Одна из его программ, названная BELLE, выиграла всемирный шахматный чемпионат компьютерных программ. В настоящее время он занят созданием новых компиляторов и новых операционных систем.

**Maurice V. Wilkes**

Морис Уилкс начал свою карьеру в 1945 г. в качестве директора Компьютерной лаборатории Кембриджского университета, где в 1949 г. он разработал первый программируемый компьютер с хранимой программой, названный Электронный Автоматический Калькулятор с Памятью на Линиях Задержки (EDSAC). Выйдя в отставку и покинув Кембридж в 1980 г., он затем стал штатным консультантом компании DEC и является ныне членом ученого совета по планированию научных исследований организации Olivetti Research Board. Уилкс является членом Британского Королевского общества с 1956 г. и в 1957 г. он был назначен первым президентом Британского Компьютерного Общества. Он получил Тьюринговскую премию в 1967 г.

Уилкс был первооткрывателем в таких областях, как машины с хранимой программой, библиотеки подпрограмм и микропрограммирование; эти концепции были использованы при создании EDSAC II. Недавно он вернулся в Кембридж и опубликовал свою автобиографию *Memoirs of a Computer Pioneer* (MIT Press).

**J. H. Wilkinson**

Джон Уилкинсон, лауреат Тьюринговской премии 1970 г., был одним из самых прославленных специалистов по численному анализу. Всю свою жизнь (он умер 5 октября 1986 г.) Уилкинсон занимался исследованиями по численному анализу, в особенности численными методами линейной алгебры и задачами теории возмущений. Его главными достижениями были создание компьютера ACE и разработка метода анализа погрешности в обратном направлении.

Уилкинсон был выпускником Тринити-Колледжа в Кембридже, Англия, и членом Королевского общества. Он возглавлял Национальную физическую лабораторию в Англии с 1946 по 1977 гг. Кроме того, каждый год, с 1977 по 1984, он три месяца проводил в качестве приглашенного профессора в Станфордском университете, где он продолжал свои исследования по обобщенной задаче на собственные значения и работал над другими проектами в области численного анализа.

### **Niklaus Wirth**

Никлаус Вирт был награжден Тьюринговской премией в 1984 г., когда он работал над проектированием и созданием персонального компьютера и однопроходного компилятора для языка Модула-2 в Швейцарском Федеральном Технологическом Институте (ETH) в Цюрихе. Проведя свой годичный отпуск в Исследовательской лаборатории компании «Ксерокс» в Пало-Альто, Калифорния, Вирт возвратился в ETH в 1986 г., где он продолжает работать над операционным обеспечением, конструкциями рабочих станций, анализом архитектур микропроцессоров и основаниями программирования.

С тех пор, как в 1963 г. Вирт получил докторскую степень по электронному машиностроению в Калифорнийском университете в Беркли, он продолжает работать над преодолением того, что он называет неоправданной сложностью конструкций компьютеров. Особую важность представляет создание им компьютера Лилит — проект, над которым он работал с 1978 г.

# Индексы классификационной схемы журнала АСМ «Computing Reviews»

Классификационная схема журнала «Computing Reviews» была разработана в течение 1979—1981 гг. международным комитетом, который возглавлял профессор Энтони Рэлстон из университета штата Нью-Йорк в Буффало. Впервые использованная в 1982 г. АСМ для классификации собственных публикаций АСМ, она приобретает все большую популярность как средство поиска литературы по информатике, относящейся к той или иной предметной области.

## **A. Литература общего характера**

### **A.0 Общий отдел**

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.

## **C. Принципы построения компьютерных систем**

### **C.0 Общий отдел**

Вирт, Никлаус. «От разработки языка программирования к созданию компьютера». С. 210.

### **C.1 Архитектуры процессоров**

#### **C.1.1 Архитектуры с одним потоком данных**

Бэкус, Джон. «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ». С. 84.

### **C.5 Реализация компьютерных систем**

#### **C.5.2 Микрокомпьютеры**

Ритчи, Дэнис М. «Размышления об исследованиях в области программного обеспечения». С. 195.

## **D. Программное обеспечение**

### **D.1 Методы программирования**

#### **D.1.1 Функциональное программирование**

Бэкус, Джон. «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ». С. 84.

#### **D.1.2 Автоматическое программирование**

Кнут, Дональд Е. «Программирование как искусство». С. 48.  
Уилкс, Морис В. «Компьютеры прежде и теперь». С. 229.

### **D.2 Разработка программного обеспечения**

#### **D.2.1 Требования/Стандарты**

Вирт, Никлаус. «От разработки языка программирования к созданию компьютера». С. 210.

#### **D.2.2 Инструменты и методы**

Флойд, Роберт. «Парадигмы программирования». С. 159.

#### **D.2.4 Проверка правильности программ**

Бэкус, Джон. «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ». С. 84.

Дейкстра, Эдсгер В. «Смиренный программист». С. 30.

#### **D.2.5 Тестирование и отладка**

Томпсон, Кен. «Размышления о том, можно ли полагаться на доверие». С. 203.

D.2.9 Управление

Кодд Э. Ф. «Реляционная база данных: практическая основа эффективности». С. 451.

D.3 Языки программирования

D.3.0 Общий отдел

Дейкстра, Эдсгер В. «Смиренный программист». С. 30.

D.3.1 Формальные определения и теория

Бэкус, Джон. «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ». С. 84.

Минский, Марвин. «Форма и содержание в информатике». С. 255.

Перлис, Алан. «Синтез алгоритмических систем». С. 16.

D.3.2 Классификация языков

Хоар, Чарлз Энтони Ричард. «Старые платья императора». С. 174.

Перлис, Алан. «Синтез алгоритмических систем». С. 16.

Томпсон, Кен. «Размышления о том, можно ли полагаться на доверие». С. 203.

D.3.3 Языковые конструкции

Дейкстра, Эдсгер В. «Смиренный программист». С. 30.

Флойд, Роберт. «Парадигмы программирования». С. 159.

Перлис, Алан. «Синтез алгоритмических систем». С. 16.

D.3.4 Процессоры

Кодд Э. Ф. «Реляционная база данных: практическая основа эффективности». С. 451.

Хоар, Чарлз Энтони Ричард. «Старые платья императора». С. 174.

Вирт, Никлаус. «От разработки языка программирования к созданию компьютера». С. 210.

Минский, Марвин. «Форма и содержание в информатике». С. 255.

D.4 Операционные системы

D.4.0 Общий отдел

Ритчи, Дэнис М. «Размышления об исследованиях в области программного обеспечения». С. 195.

D.4.1 Управление процессами

Хоар, Чарлз Энтони Ричард. «Старые платья императора». С. 174.

D.4.3 Управление системами файлов

Перлис, Алан. «Синтез алгоритмических систем». С. 16.

D.4.6 Компьютерная безопасность и защита данных

Томпсон, Кен. «Размышления о том, можно ли полагаться на доверие». С. 203.

F. Теория вычислений

F.0 Общий отдел

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.

F.1 Вычисления посредством абстрактных устройств

F.1.1 Модели вычислений

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.

Ньюэлл, Аллен и Саймон, Херберт. «Информатика как эмпирическое исследование: символы и поиск». С. 333.

F.1.2 Способы вычислений

Кук, Стивен А. «Обзор вычислительной сложности». С. 475.

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.

F.1.3 Классы сложности

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.

F.2 Анализ алгоритмов и сложность задач

F.2.0 Общий отдел

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.

F.2.1 Численные алгоритмы и задачи

Кук, Стивен А. «Обзор вычислительной сложности». С. 475.

Айверсон, Кеннет Е. «Нотация как средство мышления». С. 392.

Минский, Марвин. «Форма и содержание в информатике». С. 255.

Рабин, Микаэль. «Сложность вычислений». С. 371.

Уилкинсон Дж. «Некоторые замечания математика-вычислителя». С. 284.

F.2.2 Нечисленные алгоритмы и задачи

Рабин, Микаэль. «Сложность вычислений». С. 371.

F.3 Логика и значение программ

F.3.2 Семантика языков программирования

Скотт, Дана С. «Логика и языки программирования». С. 65.

F.4 Математическая логика и формальные языки

F.4.1 Математическая логика

Бэкус, Джон. «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ». С. 84.

Минский, Марвин. «Форма и содержание в информатике». С. 255.

Скотт, Дана С. «Логика и языки программирования». С. 65.

F.4.2 Граматики и другие переписывающие системы

Рабин, Микаэль. «Сложность вычислений». С. 371.

G. Математические вопросы теории вычислений

G.1 Численный анализ

G.1.0 Общий отдел

Рабин, Микаэль. «Сложность вычислений». С. 371.

Уилкинсон Дж. «Некоторые замечания математика-вычислителя». С. 284.

G.1.3 Численные методы линейной алгебры

Бэкус, Джон. «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ». С. 84.

Уилкинсон Дж. «Некоторые замечания математика-вычислителя». С. 284.

G.1.5 Корни и нелинейные уравнения

Бэкус, Джон. «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ». С. 84.

G.1.m Разное

Айверсон, Кеннет Е. «Нотация как средство мышления». С. 392.

G.2 Дискретная математика

G.2.1 Комбинаторика

Айверсон, Кеннет Е. «Нотация как средство мышления». С. 392.

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.

G.2.2 Теория графов

Айверсон, Кеннет Е. «Нотация как средство мышления». С. 392.

Карп, Ричард М. «Комбинаторика, сложность и случайность». С. 498.



G.3 Теория вероятностей и статистика

Кук, Стивен А. «Обзор вычислительной сложности». С. 475.

## **Н. Информационные системы**

H.2 Управление базами данных

H.2.1 Логическое проектирование

Кодд Э. Ф. «Реляционная база данных: практическая основа эффективности». С. 451.

H.2.2 Физическое проектирование

Бахман, Чарльз. «Программист-навигатор». С. 313.

H.3 Хранение и поиск информации

H.3.2 Хранение информации

Бахман, Чарльз. «Программист-навигатор». С. 313.

H.3.3 Поиск информации

Бахман, Чарльз. «Программист-навигатор». С. 313.

## **И. Методы вычислений**

I.1 Алгебраические манипуляции

I.1.1 Выражения и их представление

Айверсон, Кеннет Е. «Нотация как средство мышления». С. 392.

I.2 Искусственный интеллект

I.2.1 Приложения и экспертные системы

Уилкс, Морис В. «Компьютеры прежде и теперь». С. 229.

I.2.3 Вывод и доказательство теорем

Маккарти, Джон. «Общность в системах искусственного интеллекта». С. 299.

I.2.4 Методы и формализмы представления знаний

Маккарти, Джон. «Общность в системах искусственного интеллекта». С. 299.

I.2.6 Обучение

Маккарти, Джон. «Общность в системах искусственного интеллекта». С. 299.

I.2.7 Обработка естественного языка

Ньюэлл, Аллен и Саймон, Херберт. «Информатика как эмпирическое исследование: символы и поиск». С. 333.

I.2.8 Решение задач, методы управления и поиска

Ньюэлл, Аллен и Саймон, Херберт. «Информатика как эмпирическое исследование: символы и поиск». С. 333.

## **Ж. Применения компьютеров**

J.2 Физические науки и инженерное дело

Хэмминг Р. В. «Одна из точек зрения на информатику». С. 240.

## **К. Компьютеры и общество**

K.2 История автоматизированных вычислений

Дейкстра, Эдсгер В. «Смирный программист». С. 30.

Карп, Ричард М. «Комбинаторика. сложность и случайность». С. 498.

Ньюэлл, Аллен и Саймон, Херберт. «Информатика как эмпирическое исследование: символы и поиск». С. 333.

Уилкс, Морис В. «Компьютеры прежде и теперь». С. 229.

Уилкинсон Дж. «Некоторые замечания математика-вычислителя». С. 284.

K.3 Компьютеры и образование

K.3.0 Общий отдел

Минский, Марвин. «Форма и содержание в информатике». С. 255.

K.3.2 Компьютер и обучение информатике

Флойд, Роберт В. «Парадигмы программирования». С. 159.

Хэмминг Р. В. «Одна из точек зрения на информатику». С. 240.

K.6 Управление вычислительными и информационными системами

К.6.1 Управление проектами и людьми

Кнут, Дональд Е. «Программирование как искусство». С. 48.

Ритчи, Дэнис М. «Размышления об исследованиях в области программного обеспечения». С. 195.

К.7 Профессия программиста

К.7.0 Общий отдел

Кнут, Дональд Е. «Программирование как искусство». С. 48.

К.7.1 Профессии

Дейкстра Эдсгер В. «Смиранный программист». С. 30.

К.7.m Разное

Хэмминг Р. В. «Одна из точек зрения на информатику». С. 240.

# Именной указатель

- Адлеман (Adleman L. A.) 488, 489, 492  
Адлер (Adler I.) 518  
Айверсон (Iverson K.) 96, 226, 392  
Амман (Ammann U.) 216  
Ахо (Aho A. V.) 478
- Баккер (de Bakker J.) 72  
Барстелл (Burstall) 113  
Бахман (Bachman C.) 227, 228, 313  
Беллман (Bellman R.) 502  
Беннет (Bennett J. H.) 477, 486  
Бентам (Bentham J.) 57, 60  
Бентли (Bentley J.) 517  
Бердвуд (Beardwood) 516  
Берлинер (Berliner H.) 316  
Берклинг (Berkling K. J.) 152  
Берлекэмп (Berlekamp E. R.) 487  
Берри (Berry M. J. A.) 449  
Бем (Bohm C.) 68  
Блаув (Blaauw G. A.) 444  
Блэк (Black F. A.) 305  
Блюм (Blum M.) 382, 476, 492, 509, 519  
Бобров (Bobrow D.) 203  
Боргвард (Borgwardt K.-H.) 518, 537  
Бородин (Borodin A.) 484, 490, 493  
Боулес (Bowles K.) 216  
Брент (Brent R. P.) 384, 385  
Бринк (Brinch H.) 216  
Брукер (Brooker) 234  
Буль (Boole G.) 392  
Бургес (Burgess J.) 449  
Бэббидж (Babbage C.) 393  
Бэкус (Backus J.) 13, 84  
Бэлзер (Balzer R.) 161, 162
- Вайнберг (Weinberg S.) 472  
Вайсман (Weissman C.) 261  
Валиант (Valiant L. G.) 385, 486  
Ван Вейнгаарден (Van Wijngaarden A.) 30, 213  
Вигдерсон (Wigderson A.) 529  
Виноград (Winograd S.) 261, 383, 481, 509  
Вирт (Wirth N.) 15, 160, 188, 210
- Вожелер (Vogelaere R.) 70  
Вуджер (Woodger M.) 286
- Галлер (Galler B. A.) 333, 371  
Ганди (Gandy R.) 73  
Гелернтер (Gelernter) 163  
Гелтон (Galton F.) 270  
Гилл (Gill J.) 489, 512  
Голдстейн (Goldstine H.) 295  
Голуб (Golub) 296  
Гольдшлягер (Goldshlager L. M.) 491  
Гостлоу (Gostelow K. P. A.) 89  
Грей (Gray J. N.) 157  
Грин (Green C.) 165  
Грис (Gries D. S.) 157  
Гудвин (Goodwin C.) 286, 295
- Данциг (Dantzig G.) 296, 501, 503, 509, 511, 536  
Дейкстра (Dijkstra E. W.) 12, 29, 56, 113, 151, 160, 176  
Дейт (Date C.) 472  
Деннет (Dennett D.) 386  
Деннис (Dennies J. B.) 89  
Джолей (Jolley L. B.) 441  
Джонсон Д. (Johnson D.) 513, 514  
Джонсон С. (Johnson S.) 501  
Диффи (Diffie W.) 492  
Донахью (Donahue J. E.) 68  
Дэвис Р. (Davis R.) 163  
Дэвис П. (Davis P.) 290, 537
- Ершов А. П. 55  
Ершов Ю. Л. 73
- Зиллис (Zilles S.) 157
- Кай (Kay A.) 201  
Кайори (Cajori F.) 439  
Калтофен (Kaltofen E.) 482  
Каннинг (Canning R.) 313  
Карацуба А. А. 480  
Карлсон (Carlson W.) 159, 174, 392  
Кармаркар (Karmarkar N.) 537, 523  
Карп (Karp R. M.) 228, 387, 480, 486, 498

- Карр (Carr J.) 232  
 Карри (Curry H. B.) 69, 79  
 Кахан (Kahan W.) 296  
 Кернер (Kerner I. O.) 438, 444  
 Керниган (Kernighan B. W.) 56, 502, 514, 515  
 Кинг (King P.) 176  
 Кларк (Clark K. L.) 70  
 Клини (Kleene S.) 73  
 Кнут (Knuth D. E.) 11, 48, 172, 257, 509  
 Кобэм (Cobham A.) 477, 484  
 Ковальски (Kowalski R. A.) 78  
 Кодд (Codd E. F.) 227, 228, 451  
 Кок (Cocke J.) 162  
 Коксеттер (Coxeter H. S. M.) 55  
 Колмогоров А. Н. 478, 492  
 Копперсмит (Coppersmith O.)  
 Косински (Kosinski P.)  
 Коуэлл (Cowell D. F.) 70  
 Коэн (Cohen H.) 481, 488  
 Крайзель (Kreisel G.) 71  
 Кук Р. (Cook R.) 176, 182  
 Кук С. (Cook S. A.) 228, 257, 387, 475, 509  
 Кун (Kuhn T.) 161  
  
 Лакс (Luks E. M.) 481  
 Ландин (Landin P.) 70, 176, 186  
 Левин Л. 486, 572, 537  
 Левин М. (Levin M.) 261  
 Ледерберг (Lederberg) 359  
 Лейтон (Leighton) 517  
 Ленстра А. К. (Lenstra A. K.) 482  
 Ленстра-мл. (Lenstra H. W., Jr.) 481, 488  
 Лерер (Lehrer T.) 280  
 Леск (Lesk M.) 198  
 Лехмер Д. (Lehmer D.) 71  
 Лехмер Э. (Lehmer E.) 11, 51, 71  
 Линь Чень (Lin Shen) 502, 514, 515  
 Лифшиц (Lifschitz V.) 308  
 Ловас (Lovasz L.) 482  
 Лорие (Lorie R.) 467  
 Лоулер (Lawler E.) 509  
 Льюис (Lewis P. M.) 482  
  
 Маго (Mago G. A.) 152, 157  
 Макгеох (McGeoch) 517  
 Мак-Джонс (McJones P. R.) 132, 156, 157  
 Макдоннел (McDonnell E. E.) 443  
 Макилрой (McIlroy M. D.) 29, 196  
 Макинтайр (McIntyre D.) 445  
 Маккалок (McCulloch W. S.) 255  
 Маккарти (McCarthy J.) 71, 113, 227, 255, 300, 305, 308, 345  
 Макнотон (McNaughton R.) 508, 537  
  
 Ман (Manes E. G.) 68  
 Манна (Manna Z.) 78, 113, 132  
 Манро (Munro L.) 382, 484  
 Мартин (Martin W. A.) 265  
 Мартынюк В. В. 259  
 Матиясевич Ю. В. 537  
 Меджиддо (Megiddo N.) 518  
 Мейер (Meyer A. R.) 385, 387, 483, 537  
 Метрополис (Metropolis N.) 70  
 Микали (Micali S.) 492, 493  
 Миллер Г. (Miller G.) 481  
 Миллер Р. (Miller R.) 509, 530  
 Милль (Mill J. S.) 51  
 Милн (Milne R.) 72, 73  
 Милнер (Milner R.) 73, 113  
 Минский (Minsky M.) 165, 168, 225, 255, 268  
 Миттман (Mittman B.) 172  
 Мозес (Moses J.) 265, 266  
 Моррис Б. (Morris B.) 196  
 Моррис Дж. (Morris J. H.) 113, 126, 157, 234  
 Мостовски (Mostowski A.) 73  
 Моцкин (Motzkin T.) 382  
 Мочли (Mauchly J.) 230  
 Мюллер (Mueller R. E.) 54  
  
 Наур (Naur P.) 37, 85, 157, 176  
 Нильсон (Nilsson N. J.) 357  
 Норман (Norman D.) 196  
 Ньюэлл (Newell A.) 226, 251, 302, 333  
  
 Олворд (Alvord L.) 449  
 Орт (Orth D. L.) 435, 444  
 Осана (Ossanna J.) 196  
 Офман Ю. П. 480  
 Охрэн (Ohran R.) 218  
  
 Парк (Park D.) 73  
 Парнас (Parnas D.) 160, 534  
 Патерсон (Paterson M.) 383, 385  
 Пейперт (Papert S.) 225, 255, 258, 267, 272  
 Пенроуз (Penrose R.) 72  
 Перлис (Perlis A.) 11, 12, 16, 48, 234  
 Пиаже (Piaget J.) 269, 369  
 Пим (Pym J.) 178  
 Пиппенджер (Pippenger N.) 490  
 Пирл (Pearl J.) 367  
 Питтс (Pitts W.) 255  
 Платек (Platek R.) 71  
 Плодджер (Praeger P. J.) 56  
 Плоткин (Plotkin G.) 73  
 Полей (Poley S.) 56  
 Пост (Post E.) 343, 507  
 Пратт (Pratt V. R.) 490

Пресбургер (Presburger M.) 377

Путман (Putman) 537

Рабин (Rabin M.) 13, 66, 69, 228, 371, 387, 471, 476, 482, 487, 537

Райвест (Rivest R. L.) 488, 492

Раков (Rackoff C.) 493

Рейнольдс (Reynolds J. D.) 73, 113, 151, 157

Ритчи Д. (Ritchie D.) 14, 195, 203, 477

Ритчи Р. (Ritchie R. W.) 477

Робинсон Дж. (Robinson J.) 69, 509, 537

Робинсон Р. (Robinson R.) 69

Роджерс (Rogers H.) 508

Розен (Rosen B. K.) 156

Розенберг (Rosenberg A.) 509

Рот (Roth J. P.) 500

Савидж (Savage J. E.) 477

Саймон (Simon H. A.) 172, 226, 215, 302, 333

Саппес (Suppes P.) 71

Селфридж (Selfridge O.) 255

Семмет (Sammet J. E.) 84

Скотт (Scott D. C.) 13, 65, 87, 113, 132, 263, 371

Слэгл (Slagle J. R.) 266

Смэйл (Smale S.) 479, 509, 537

Сноу (Snow C. P.) 55

Соловей (Solovay R.) 481, 487, 509, 519

Соломонофф (Solomonoff R.) 255

Спайзер (Spieser A. P.) 212

Спенс (Spence R.) 449

Стил (Steel T.) 71

Стирнс (Stearns R. E.) 476, 482, 508, 537

Стой (Stoy J. E.) 68

Стокмейер (Stockmeyer L. J.) 483, 491, 517, 519, 537

Стречи (Strachey C.) 66, 71, 72, 186

Сэмюэл (Samuel) 359

Тарский (Tarski A.) 69

Теннет (Tennet R.) 67

Томпсон (Thompson K.) 14, 195, 202, 203

Тоом А. Л. 257, 480

Тоули (Tolle D.) 157

Троттер (Trotter H.) 70

Тьюринг (Turing A.) 73, 187, 224, 229, 231, 234, 240, 255, 268, 285, 343, 363, 383, 536

Уайтхед (Whitehead A. N.) 393

Уатсон (Watson J. D.) 200

Уилкинсон (Wilkinson J. H.) 224, 227, 284

Уилкс (Wilkes M.) 224, 229, 284

Уильямс (Williams J. H.) 157

Ульман (Ullman J. D.) 478

Упфал (Upfal E.) 529

Успенский В. А. 478, 495

Фадлеева В. Н. 292

Фалкерсон (Fulkerson R.) 501, 503, 537

Фалкофф (Falkoff A. D.) 445

Фейгенбаум (Feigenbaum E. A.) 268, 359

Фейджин (Fagin R.) 157

Фейнман (Feynman R. P.) 256, 278

Фелдман (Feldman J.) 268

Феферман (Feferman S.) 71

Фишер (Fischer M. J.) 56, 385, 387, 483

Флойд (Floyd R. W.) 13, 113, 159, 187, 386

Фокс (Fox L.) 73, 286, 289, 295

фон Нейман (Von Neumann J.) 230, 290, 295, 479

фон цур Гатен (Von zur Gathen J.) 493

Форд (Ford L.) 503, 537

Форсайт (Forsythe G.) 172, 241

Франк (Fronk G. A.) 157

Френкель (Frenkel Karen A.) 522, 525

Фридберг (Fridberg R. M.) 301, 302

Хаммерсли (Hammersley) 516

Харрисон (Harrison M.) 509

Хартманис (Hartmanis J.) 476, 482, 508, 537

Хаски (Huskey H.) 70, 212, 235, 286, 289

Хачиян Л. Г. 479, 481

Хевит (Hewitt C.) 261

Хелд (Held M.) 501

Хелман (Hellman M. E.) 492

Хилмор (Hillmore J.) 178

Хоар (Hoare C. A. R.) 14, 74, 113, 174, 216

Холтон (Halton) 516

Хопкрофт (Hopcroft J. E.) 478, 509

Хофман (Hoffman A.) 509

Хофманн (Hofmann K. H.) 73

Хэзони (Hazony Y.) 449

Хэйес (Hayes P. J.) 305

Чайтин (Chaitin G. J.) 492

Чёрч (Church A.) 69, 73, 79, 343

- Шамир А.** (Shamir A.) 488, 492  
**Шамир Р.** (Shamir R.) 518  
**Шварц** (Schwartz J. T.) 488, 489  
**Шеклтон** (Shackleton P.) 175  
**Шеннон** (Shannon C. E.) 257, 342, 477  
**Шёнфинкель** (Schönfinkel) 130  
**Шёнхаге** (Schönhage) 385, 480  
**Шортлифф** (Shortliffe E. H.) 163  
**Шоу Дж.** (Show J. C.) 333, 334  
**Шоу Р.** (Shaw R. A.) 491  
**Шрауб** (Shrobe H.) 170  
**Штрассен** (Strassen V.) 258, 384, 385
- Шэдуэлл** (Shadwell T.) 67  
**Эдмондс** (Edmonds J.) 486, 505, 537  
**Эйленберг** (Eilenberg S.) 70  
**Экерт** (Eckert P.) 230  
**Эктон** (Acton F.) 70  
**Эрвинд** (Arvind) 89  
**Эттниви** (Attneave F.) 272  
**Эшенхёрст** (Ashenhurst R.) 10, 70  
**Ямада** (Yamada H.) 477, 508, 537  
**Яо** (Yao A. C.) 492

# Предметный указатель

- Автоматизация** программирования (automatic programming) 52, 162, 232
- Ада** (Ada) 25, 192, 221
- алгебра программ** (algebra of programs) 86, 100, 113, 128, 155
- **ФП-систем** 113
- Алгол** (Algol) 16, 37, 141, 176, 188, 214, 438
- алгоритмы** (algorithms) 17, 160, 295, 378, 479
- **вероятностные** (probabilistic ~) 388, 487, 515
- **рандомизированные** (randomized ~) 316, 519, 537
- **хорошие** (good ~) 522, 537
- аппаратная часть** (hardware) 31, 152, 219, 469
- аппликативные модели** (applicative models) 88
- **системы** (~ systems) 92
- атом** (atom) 104
- База данных** (database) 300, 305 313, 330, 373, 386, 453, 469
- **реляционная** (relational) 452—474
- **знаний** (knowledge base) 330, 368, 369
- бактериальная теория** (germ theory) 338
- блочное кодирование** (block-encoding) 390
- Бэкуса—Наура форма** (БНФ) (Backus-Naur form (BNF)) 37, 42, 84
- Ввода-вывода прерывания** (input/output interrupts) 33
- вероятностный анализ** (probabilistic analysis) 521, 524, 537
- весовые функции** (cost functions) 373, 390
- внутреннее произведение** (inner product) 92, 428, 433
- время вычислений** (computation time) 258, 477
- вход в блок** (entry to a block) 21
- выполнимости проблема** (satisfiability problem) 510, 524
- вычислений теория** (computational theory) 66, 256, 371, 475, 521
- вычислимости теория** (computability theory) 506
- вычисляемые функции** (computable functions) 374
- вычислительная машина** (automatic computers) 31
- **аналоговая** (analog computer) 247
- **сложность см. сложность**
- вычислительные компромиссы** (computational trade-offs) 258
- Гауссово исключение** (gaussian elimination) 171, 293, 295
- Гильберта десятая проблема** (Hilbert's Tenth problem) 507, 522, 536
- гипотеза о символьной системе** (symbol system hypothesis) 341
- граф** (graph) 423, 481
- Данных структуры** (data structures) 24, 234, 300, 389
- двухместная транспозиция** (dyadic transpose) 433, 438
- деревья поиска** (search trees) 355
- детализация** (subordination of detail) 399
- диалоговое программирование** (conversational programming) 21, 29
- динамическое программирование** (dynamic programming) 162
- дистрибутивность** (distributivity) 430
- доставка** (fetching) 137
- Задача о коммивояжере** (traveling salesman problem) 501, 522, 545
- закон больших чисел** (large number law) 516
- запоминание** (storing) 137

защищенные линии связи (secure communications) 390

**Игровые программы** (game-playing programs) 356, 362, 369

идемпотентность (idempotency) 127

иерархические системы (hierarchical systems) 44, 69

изменяемые части (changeable parts) 95

изоморфизм (isomorphism) 80

инженер знаний (knowledge engineer) 369

интеллектуальность (intelligence) 357

интеллектуальные системы (intelligent systems) 364

интерпретация (interpretation) 339

интерференция (interference) 324

информатика (informatics) 267, 273, 525, 535

информации теория (information theory) 342, 361, 492

информация (information) 353, 361

исключительные условия (exception conditions) 143

искусственный интеллект (ИИ) (artificial intelligence) 52, 164, 236, 300, 336, 350, 359, 364

историческая чувствительность (history sensitivity) 88, 92

итерация (iteration) 124

**Кластеризация** (clustering) 319

клеточная теория (cell doctrine) 337

когнитивная психология (cognitive psychology) 346

комбинаторика (combinatorics) 500

комбинаторная оптимизация (combinatorial optimization) 511, 536

комбинаторный взрыв (exponential explosion) 357, 522

комбинаторы (combinators) 79, 130

комбинационные формы (combining forms) 86, 95, 154

компилятор (compiler) 176, 214, 221, 263, 468

— однопроходный (single-pass) 177

компьютерная арифметика (computer arithmetic) 375

— графика (~ graphics) 236

компьютерные преступления (~ crime) 290

компьютерология (computerology) 290

конвейерная организация (pipelining) 29

конечные автоматы (state-machine) 170

контекст (context) 309

контекстно-свободные грамматики (context-free grammars) 69, 260, 375

контроль доступа (access control) 209

корректность программ (correctness of programs) 41, 52, 98, 113, 123

Кука теорема (Cook theorem) 537

**Лилит** (Lilith) 211, 219

линейное программирование (linear programming) 293, 481, 502, 517, 537

Лисп (Lisp) 28, 37, 88, 92, 131, 165, 261, 345

логическое программирование (logic programming) 28, 303, 522

**Марьяжа проблема** (marriage problem) 504, 522

массивы (arrays) 399, 439

математическая нотация (mathematical notation) 393, 422, 438, 442, 449

матричные вычисления (matrix computation) 295, 481

Меза (Mesa) 217

метод ветвей и границ (branch-and-bound method) 161, 502

— описаний (circumscription) 308

— «разделяй и властвуй» (divide-and-conquer) 161, 171

методы доступа (access methods) 316, 478

— «слабые» и «сильные» («weak» and «strong») 358

модели потоков данных (data flow models) 152

— предприятия (enterprise models) 330

— произвольного доступа (random access ~) 478

монитор (monitor) 186

мониторинг (monitoring) 26

Мультикс (Multics) 183, 197

мультипрограммный режим (multiprogramming) 216, 322

мышь (mouse) 218

**Надежность** (reliability) 38, 176, 192, 390

недетерминированное программирование (nondeterministic programming) 164

нелинейные уравнения (nonlinear equations) 126

немонотонность (nonmonotonicity) 307

неразрешимость (undecidability) 522, 536



- Ньютона симметричные функции (Newton's symmetric functions) 432
- Обозначение (designation) 339, 344
- Общий решатель проблем (General Problem Solver) 301, 347, 358
- объектно-ориентированное программирование (object-oriented programming) 28
- объектный код (object code) 177
- объекты (objects) 102, 339
- операторы (operators) 440
- операционная система (operating system) 183, 195
- операционные модели (operational models) 88
- описание процедуры (procedure declaration) 22
- оптимальное решение (optimal solution) 528, 537
- остановки проблема (halting problem) 507, 522, 536
- отладка (debugging) 38, 208
- «отпечатков пальцев» функция («fingerprinting» function) 520
- оценки времени (time bounds) 479
- произведения времени и памяти (time-space product bounds) 484
- стоимости решения (bound on cost of solution) 537
- Память (memory) 258
- параллельная работа (parallel operation) 26
- параллельные вычисления (parallel computations) 259, 489, 524
- процессоры ( $\sim$  processors) 530
- Паскаль (Pascal) 188, 193, 210, 216, 220
- переменные (variables) 20
- перестановки (permutations) 419
- персональный компьютер (personal computer) 217
- погрешности округления (rounding errors) 284, 293
- подпрограмма (subroutine) 16, 35
- поиск в решении задач (search in problem solving) 353
- с выбором первой лучшей вершины (best-first search) 358
- поисковые системы (retrieval systems) 317, 361, 368
- полиномиальная сводимость (polynomial reducibility) 524, 537
- полиномы (polynomials) 382, 408, 416, 435
- помещение на стек (push) 138
- потоки в сетях (network flow) 503
- почти оптимальные решения (near-optimal solutions) 537
- правила перехода (transition rules) 142
- правило метакомпозиции (metacomposition rule) 135
- практически невыполнимые задачи (intractable problems) 386
- представления (representations) 362, 414, 444
- знаний ( $\sim$  of knowledge) 300
- синтаксиса (syntax  $\sim$ ) 24
- функций (of functions) 414
- Пресбургера арифметика (Presburger's arithmetic) 377, 387, 483
- проблемное пространство (problem space) 353, 363
- программное обеспечение (software) 33, 39, 161, 219
- производственные системы (production systems) 301
- проектирование компьютеров (computer design) 152, 236
- Пролог (Prolog) 28, 304
- простых чисел распознавание 481
- прямой доступ (direct access) 318, 323
- Разбиения (partitioning) 428
- разделение времени (time sharing) 21, 230
- разложение (expansion) 121, 128
- разрезы данных (views) 465
- рандомизация (randomizing) 316
- раскрутка (bootstrapping) 220, 267
- распознавание образов (pattern recognition) 238
- распределение (distribution) 429
- распределенная связь (distributed communications) 531
- распределенные протоколы ( $\sim$  protocols) 531
- рекурсии теорема (recursion theorem) 122, 257
- рекурсия (recursion) 176, 257, 381
- реляционная алгебра (relational algebra) 458
- реляционное исчисление ( $\sim$  calculus) 451
- реляционные модели ( $\sim$  models) 454
- решение задач (problem solving) 351
- решетки (lattices) 74
- Сводимость (reducibility) 507, 522
- семантика (semantics) 56, 88, 108, 133
- аксиоматическая (axiomatic  $\sim$ ) 98, 186

- денотационная (denotational ~) 97
- неподвижной точки (fixpoint ~) 78
- семантические структуры (~ structures) 74
- символьная логика (symbolic logic) 427
- структура (~ structure) 300, 339, 345
- символы (symbols) 300, 333, 336
- синтаксис (syntax) 24, 234, 260, 405
- синтаксический анализ (parsing) 159, 373, 381
- симплекс-метод (simplex method) 503, 518, 536
- системная программа (system program) 144
- системы счисления (number system) 415
- ситуационное исчисления (situation calculus) 306
- сложение (summarization) 384, 429
- сложности классы (complexity classes) 537
- мера (~ measure) 477
- теория (~ theory) 65, 373, 457, 498, 524, 533, 537
- сложность алгоритмов (~ of algorithms) 476
- Смолток (Smalltalk) 28
- Снобол (Snobol) 260
- снятие со стека (pop) 138
- собственные значения (eigenvalues) 292
- совместный доступ (shared access) 322
- сокрытие информации принцип (information hiding) 161
- сопоставление с образцом (pattern matching) 524
- сортировка (sorting) 171, 318, 376, 386
- быстрая (quicksort) 174
- списки (lists) 29, 344, 373, 389
- стиль программирования (style in programming) 56, 85, 100, 250
- структурное программирование (structured programming) 160, 166
- стягивающее дерево (spanning tree) 424
- Тарского семантика (Tarski's semantics) 69
- тектоника плит (plate tectonics) 338
- теория автоматов (automata theory) 69
- потоков в сетях (network flow theory) 321
- тестирование (testing) 41
- транзакция (transaction) 21
- трансляция (translation) 24
- Тьюринга машина (Turing machine) 16, 88, 153, 324, 536
- Умножение больших чисел 480
- матриц 109, 119
- универсальная реляционность (uniform relational property) 461
- уровней абстракции метод 161
- условные выражения (conditional expressions) 106
- уточнения проблема (qualification problem) 307
- Физическая символьная система 333, 336, 339, 364, 367
- фон Неймана «узкое место» (von Neumann «bottleneck») 89, 141, 110
- формальная логика (formal logic) 341
- формальные системы функционального программирования 100, 132, 156
- Фортран (Fortran) 36, 43, 84, 172, 179, 181
- функциональное программирование 28, 85, 92, 108, 153
- пространство (function space) 72, 78
- системы переходов состояний (ФСПС) (applicative state transition system (AST systems)) 88, 92, 100, 140, 156
- функциональные формы (functional forms) 93, 101, 105, 117
- Фурье быстрое преобразование (БПФ) (fast Fourier transform) (FFT) 373, 384, 480
- Хвост (tail) 104
- Хомского иерархия (Chomsky hierarchy) 69
- хэшинг (hashing) 316, 468
- Цифровой компьютер (digital computer) 342
- Частичные значения (partial values) 74
- функции (~ functions) 74

- Чёрча—Россера теорема (Church-Rosser theorem) 131  
 численный анализ (numerical analysis) 252, 284, 289, 292  
 чистка (purge) 138
- Шахматные программы** (chess programs) 355, 358, 369  
*Шелла* сортировка (Shell sort) 175
- Эвристика (heuristic) 523, 527  
 эвристический поиск (~ search) 333, 347, 364, 134  
 эвристическое программирование (~ programming) 259, 534  
 Эйлер (Euler) 210, 213  
 экспертные системы (expert systems) 330, 367
- эмпирические основания (empirical base) 365  
 Эниак (ENIAC) 5, 229
- Язык Си (C language) 195  
 — запросов на примерах (Query-by-Example (QBE)) 461, 468, 471  
 — обработки данных (data manipulation language) (DML)) 457  
 — определения данных (data structure definition language) 457  
 языки программирования (programming languages) 17, 42, 62, 95, 110, 260, 393, 397, 399, 458, 470  
 — — высокого и низкого уровня (high and low level ~) 234  
 — — контекстно-свободные (context free ~) 70, 175, 210  
 — — разработка 168, 175, 210  
 — — хорошие и плохие 232  
 ячейки (cells) 137

## СОДЕРЖАНИЕ

От редактора перевода . . . . .	5
Предисловие к русскому изданию . . . . .	8
Предисловие . . . . .	9
 ВВЕДЕНИЕ К ЧАСТИ I. Языки и системы программирования (С. Л. Грэхем). Перевод С. В. Чудова . . . . .	11
А. Дж. Перлис (1966). Синтез алгоритмических систем. Перевод В. В. Мартынюка . . . . .	16
Эдсгер В. Дейкстра (1972). Смиранный программист. Перевод А. А. Бря- динской . . . . .	30
Дональд Е. Кнут (1974). Программирование как искусство. Перевод В. В. Мартынюка . . . . .	48
Дана С. Скотт (1976). Логика и языки программирования. Перевод Е. В. Келлиной . . . . .	65
Дж. Бэкус (1977). Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ. Перевод В. В. Мартынюка . . . . .	84
Роберт Флойд (1978). Парадигмы программирования. Перевод М. В. Ха- тунцевой . . . . .	159
Чарлз Энтони Ричард Хоар (1980). Старые платья императора. Перевод А. А. Брядинской . . . . .	174
Дэнис М. Ритчи (1983). Размышления об исследованиях в области про- граммного обеспечения. Перевод Н. Б. Дерябина . . . . .	195
Кен Томпсон (1983). Размышления о том, можно ли полагаться на дове- рие. Перевод Н. Б. Дерябина . . . . .	203
Никлаус Вирт (1984). От разработки языка программирования к созда- нию компьютера. Перевод М. В. Хатунцевой . . . . .	210
 ВВЕДЕНИЕ К ЧАСТИ II. Компьютеры и методы их использования (Р. Л. Эшенхёрст). Перевод С. В. Чудова . . . . .	224
М. В. Уилкс (1967). Компьютеры прежде и теперь. Перевод В. В. Мар- тынюка . . . . .	229
Р. В. Уэлминг (1968). Одна из точек зрения на информатику. Перевод Х. Д. Икрамова . . . . .	240
М. Минский (1969). Форма и содержание в информатике. Перевод В. В. Мартынюка . . . . .	255
Дж. Уилкинсон (1970). Некоторые замечания математика-вычислителя. Перевод Х. Д. Икрамова . . . . .	284
Джон Маккарти (1971). Общность в системах искусственного интел- лекта. Перевод В. Г. Толстоусовой . . . . .	299
Чарльз В. Бахман (1973). Программист-навигатор. Перевод В. Г. Тол- стоусовой . . . . .	313
Аллен Ньюэлл, Херберт Саймон (1975). Информатика как эмпирическое исследование: символы и поиск. Перевод С. В. Чудова . . . . .	333

Микаэль О. Рабин (1976). Сложность вычислений. Перевод Н. А. Карповой	371
К. Е. Айверсон (1979). Нотация как средство мышления. Перевод В. В. Мартынюка	392
Э. Ф. Кодд (1981). Реляционная база данных: практическая основа эффективности. Перевод В. Г. Толстоусовой	451
Стивен А. Кук (1982). Обзор сложности вычислений. Перевод Н. А. Карповой	475
Ричард М. Карп (1985). Комбинаторика, сложность и случайность. Перевод Н. А. Карповой	498
Карен Френкель. Постскрипtum. Сборка теории сложности из кусков. Перевод Н. А. Карповой	522
Карен Френкель. Постскрипtum. Интервью тьюринговских лауреатов. Сложность и параллельные вычисления. Интервью с Ричардом Карпом. Перевод Н. А. Карповой	525
Биографии авторов. Перевод С. В. Чудова	538
Индексы классификационной схемы журнала АСМ «Computing Reviews». Перевод С. В. Чудова	545
Именной указатель	550
Предметный указатель	554

### Научное издание

*А. Дж. Перлис, Эдсгер В. Дейкстра,  
Дональд Е. Кнут и др.*

### ЛЕКЦИИ ЛАУРЕАТОВ ПРЕМИИ ТЬЮРИНГА

Заведующий редакцией чл.-корр. АН СССР  
В. И. Арнольд  
Зам. зав. редакцией А. С. Попов  
Ст. научн. ред. С. В. Чудов  
Редактор Н. С. Полякова  
Художник Л. М. Муратова  
Художественный редактор В. И. Шаповалов  
Технический редактор И. И. Володина  
Корректор В. И. Киселева

ИБ № 7687

Сдано в набор 23.06.92. Подписано к печати 11.02.93. Формат 60×88<sup>1</sup>/<sub>16</sub>. Бумага типографская № 1. Печать высокая. Гарнитура литературная. Объем 17,5 бум. л. Усл. печ. л. 34,3. Усл. кр.-отт. 34,3. Уч.-изд. л. 36,67. Изд. № 1/8103. Тираж 2000 экз. Зак. 467. С052.

Издательство «Мир»  
Министерства печати и информации  
Российской Федерации  
129820, ГСП, Москва, 1-й Рижский пер., 2.

Московская типография № 11  
Министерства печати и информации  
Российской Федерации  
113105, Москва, Нагатинская ул., д. 1

