APL (programming language)

APL

Paradigm	array, functional, structured, modular
Appeared in	1964
Designed by	Kenneth E. Iverson
Developer	Kenneth E. Iverson
Typing discipline	dynamic
Major implementations	IBM APL2, Dyalog APL, APL2000, Sharp APL, APLX
Dialects	A+, Dyalog APL, APLNext
Influenced by	mathematical notation
Influenced	J,[1] K,[2] Mathematica, MATLAB,[3] Nial,[4] PPL, Q

APL (named after the book *A Programming Language*)[5] is an interactive array-oriented language and integrated development environment which is available from a number of commercial and non-commercial vendors[6] and for most computer platforms.[7] It is based on a mathematical notation developed by Kenneth E. Iverson.

APL has a combination of unique and relatively uncommon features that appeal to programmers and make it a productive programming language:[8]

- It is concise, using symbols rather than words and applying functions to entire arrays without using explicit loops.
- It is solution focused, emphasizing the expression of algorithms independently of machine architecture or operating system.
- It has just one simple, consistent, and recursive precedence rule: the right argument of a function is the result of the entire expression to its right.
- It facilitates problem solving at a high level of abstraction.

APL is used in scientific,[9] actuarial,[8] statistical,[10] and financial applications where it is used by practitioners for their own work and by programmers to develop commercial applications. It was an important influence on the development of spreadsheets, functional programming,[11] and computer math packages.[3] It has also inspired several other programming languages.[1] [2] [4] It is also associated with rapid and lightweight development projects in volatile business environments.[12]

History

The first incarnation of what was later to be the APL programming language was published and formalized in *A Programming Language*,[5] a book describing a notation invented in 1957 by Kenneth E. Iverson while at Harvard University. Iverson had developed a mathematical notation for manipulating arrays that he taught to his students.

In 1960, he began work for IBM and working with Adin Falkoff, created APL based on the notation he had developed. This notation was used inside IBM for short research reports on computer systems, such as the Burroughs B5000 and its stack mechanism when stack machines versus register machines were being evaluated by IBM for upcoming computers.

Also in 1960, Iverson was already also using his notation in a draft copy of Chapter 6 called "A programming language" for the book he was writing with Fred Brooks, *Automatic Data Processing*, which would later be published in 1963.[13] [14]

Published in 1962, the notation described in *A Programming Language* was recognizable yet distinct from later APL.

As early as 1962, the first attempt to use the notation to describe a complete computer system happened after Falkoff discussed with Dr. William C. Carter his work in the standardization of the instruction set for the machines that later became the IBM System/360 family.

In 1963, Dr. Herbert Hellerman, working at the IBM Systems Research Institute, implemented a part of the notation on an IBM 1620 computer, and it was used by students in a special high school course on elementary functions. This implementation of a portion of the notation was called PAT (Personalized Array Translator).[15]

In 1963, Falkoff, Iverson, and Edward H. Sussenguth Jr., all working at IBM, used the notation for a formal description of the IBM System/360 series machine architecture and functionality which resulted in a paper published in IBM Systems Journal in 1964. After this was published, the team turned their attention to an implementation of the notation on a computer system. One of the motivations for this focus of implementation was the interest of John L. Lawrence who had new duties with Science Research Associates, an educational company bought by IBM in 1964. Lawrence asked Iverson and his group to help utilize the language as a tool for the development and use of computers in education.[16]

After Lawrence M. Breed and Philip S. Abrams of Stanford University joined the team at IBM Research, they continued their prior work on an implementation programmed in FORTRAN IV for a portion of the notation was done for the IBM 7090 computer running under the IBSYS operating system. This work was finished in late 1965 and later known as IVSYS (Iverson System). The basis of this implementation was described in detail by Abrams in a Stanford University Technical Report, "An Interpreter for Iverson Notation" in 1966.[17] Like Hellerman's PAT system earlier, this implementation did not include the APL character set but used special English reserved words for functions and operators. The system was later adapted for a time-sharing system and, by November 1966, it had been reprogrammed for the IBM/360 Model 50 computer running in a time sharing mode and was used internally at IBM.[18]

IBM typeballs (one OCR) with clip, €2 coin for scale

A key development in the ability to use APL effectively, before the widespread use of CRT terminals, was the development of a special IBM Selectric typewriter interchangeable typeball with all the special APL characters on it. This was used on paper printing terminal workstations using the Selectric typewriter and typeball mechanism, such as the IBM 1050 and IBM 2741 terminal. Keycaps could be placed over the normal keys to show which APL characters would be entered and typed when that key was struck. For the first time, a programmer could actually type in and see real APL characters as used in Iverson's notation and not be forced to use awkward English keyword representations of them. Falkoff and Iverson had the special APL Selectric typeballs, 987 and 988, designed in late 1964, although no APL computer system was available to use them.[19] Iverson cited Falkoff as the inspiration for the idea of using an IBM Selectric typeball for the APL character set.[20]

The IBM 2741 keyboard layout with the APL typeball print head inserted looked this way to the programmer:

Some APL symbols, even with the APL characters on the typeball, still had to be typed in by over-striking two existing typeball characters. An example would be the "grade up" character which had to be made from a "delta" (shift-H) and a "Sheffer stroke" (shift-M). This was necessary because the APL character set was larger than the 88 characters allowed on the Selectric typeball.

The first APL interactive login and creation of an APL workspace was in 1966 by Larry Breed using a 1050 terminal at the IBM Mohansic Labs near Thomas J. Watson Research Center, the home of APL, in Yorktown Heights, New York.[19]

IBM was chiefly responsible for the introduction of APL to the marketplace.

APL was first available in 1967 for the IBM 1130 as *APL\1130*.[21] [22] It would run in as little as 8k 16 bit words of memory, and used a dedicated 1 megabyte hard disk.

APL gained its foothold on mainframe timesharing systems from the late 1960s through the early 1980s. Additional improvements in performance for selected IBM 370 mainframe systems included the "APL Assist Microcode" in which some support for APL execution was included in the actual firmware as opposed to APL being exclusively a software product. Somewhat later, as suitably performing hardware was finally becoming available in the mid to late-1980s, many users migrated their applications to the personal computer environment.

Early IBM APL interpreters for IBM 360 and IBM 370 hardware implemented their own multi-user management instead of relying on the host services, thus they were timesharing systems in their own right. First introduced in 1966, the *APL\360*[23] [24] [25] system was a multi-user interpreter. The ability to programmatically communicate with the operating system for information and setting interpreter system variables was done through special privileged "I-beam" functions, using both monadic and dyadic operations.[26]

In 1973, IBM released *APL.SV* which was a continuation of the same product, but which offered shared variables as a means to access facilities outside of the APL system, such as operating system files. In the mid 1970s, the IBM mainframe interpreter was even adapted for use on the IBM 5100 desktop computer, which had a small CRT and an APL keyboard, when most other small computers of the time only offered BASIC. In the 1980s, the *VSAPL* program product enjoyed widespread usage with CMS, TSO, VSPC, MUSIC/SP and CICS users.

In 1973-1974, Dr. Patrick Haggerty introduced the APL interpreter on Sperry Corporation computers. Dr. Haggerty, single-handedly wrote the University of Maryland APL processor for the Sperry Univac 1100 Series mainframe computers. At the time, Sperry had nothing. In 1974, student Alan Stebbens was assigned the task of implementing an internal function. And student Bill Linton caused massive dumps to occur as he practiced developing APL programs in the third-floor TTY room, causing Dr. Haggerty to burst through the TTY door to halt the practice until Dr. Haggerty fixed the APL interpreter bug.

Several timesharing firms sprang up in the 1960s and 1970s which sold APL services using modified versions of the IBM APL\360[25] interpreter. In North America, the better-known ones were I. P. Sharp Associates, Scientific Time Sharing Corporation, and The Computer Company (TCC). With the advent first of less expensive mainframes such as the IBM 4331 and later the personal computer, the timesharing industry had all but disappeared by the mid 1980s.

Sharp APL was available from I. P. Sharp Associates, first on a timesharing basis in the 1960s, and later as a program product starting around 1979. *Sharp APL* was an advanced APL implementation with many language extensions, such as *packages* (the ability to put one or more objects into a single variable), file system, nested arrays, and shared variables.

APL interpreters were available from other mainframe and mini-computer manufacturers as well, notably Burroughs, CDC, Data General, DEC, Harris, Hewlett-Packard, Siemens, Xerox, and others.

In 1979, Iverson received the Turing Award for his work on APL.[27]

APL2

Starting in the early 1980s, IBM APL development, under the leadership of Dr Jim Brown, implemented a new version of the APL language which contained as its primary enhancement the concept of *nested arrays*, where an array can contain other arrays, as well as new language features which facilitated the integration of nested arrays into program workflow. Ken Iverson, no longer in control of the development of the APL language, left IBM and joined I. P. Sharp Associates, where one of his major contributions was directing the evolution of Sharp APL to be more in accordance with his vision.

As other vendors were busy developing APL interpreters for new hardware, notably Unix-based microcomputers, APL2 was almost always the standard chosen for new APL interpreter developments. Even today, most APL vendors cite APL2 compatibility, which only approaches 100%, as a selling point for their products.

APL2 for IBM mainframe computers is still available today, and was first available for CMS and TSO in 1984.[28] The APL2 Workstation edition (Windows, OS/2, AIX, Linux, and Solaris) followed much later in the early 1990s.

Microcomputers

The first microcomputer implementation of APL was on the 8008-based MCM/70, the first general purpose personal computer, in 1973.

IBM's own IBM 5100 microcomputer (1975) offered APL as one of two built-in ROM-based interpreted languages for the computer, complete with a keyboard and display that supported all the special symbols used in the language.

The VideoBrain Family Computer, released in 1977, only had one programming language available for it and that was a dialect of APL called APL/S.[29]

A Small APL for the Intel 8080 called EMPL was released in 1977, and Softronics APL, with most of the functions of full APL, for 8080-based CP/M systems was released in 1979.

In 1977, the Canadian firm Telecompute Integrated Systems, Inc. released a business-oriented APL interpreter known as TIS APL, for Z80-based systems. It featured the full set of file functions for APL, plus a full screen input and switching of right and left arguments for most dyadic operators by introducing ~. prefix to all single character dyadic functions such as - or /.

Vanguard APL was available for Z80 CP/M-based processors in the late 1970s. TCC released APL.68000 in the early 1980s for Motorola 68000-based processors, this system being the basis for MicroAPL Limited's APLX product. I. P. Sharp Associates released a version of their APL interpreter for the IBM PC and PC/370[30] - for the IBM PC, an emulator was written which facilitated reusing much of the IBM 370 mainframe code. Arguably, the best known APL interpreter for the IBM Personal Computer was STSC's APL*Plus/PC.

The Commodore SuperPET, introduced in 1981, included an APL interpreter developed by the University of Waterloo.

In the early 1980s, the Analogic Corporation developed *The APL Machine*, which was an array processing computer designed to be programmed only in APL. There were actually three processing units, the user's workstation, an IBM PC, where programs were entered and edited, a Motorola 6800 processor which ran the APL interpreter, and the Analogic array processor which executed the primitives. At the time of its introduction The APL Machine was likely

the fastest APL system available. Although a technological success, The APL Machine was a marketing failure. The initial version supported a single process at a time. At the time the project was discontinued, the design had been completed to allow multiple users. As an aside, an unusual aspect of The APL Machine was that the library of workspaces was organized such that a single function or variable which was shared by many workspaces existed only once in the library. Several of the members of The APL Machine project had previously spent a number of years with Burroughs implementing *APL\700*.

At one stage, Microsoft Corporation planned to release a version of APL, but these plans never materialized.

An early 1978 publication of Rodnay Zaks from Sybex was *A microprogrammed APL implementation* ISBN 0895880059 which is the complete, total source listing for the microcode for a Digital Scientific Corporation Meta 4 microprogrammable processor implementing APL. This may have been the substance of his PhD thesis.

Extensions

Recent extensions to APL include:

- Object-oriented programming[31]
- Support for .Net,[31] ActiveX,[32] operating system resources & connectivity
- APL as a native .NET language using Visual Studio 2008[33]
- Integrated charting[34] and manipulation of SQL databases
- XML-array conversion primitives[31] [35]
- Lambda expressions [36][37]

Overview

Over a very wide set of problem domains (math, science, engineering, computer design, robotics, data visualization, actuarial science, traditional DP, etc.) APL is an extremely powerful, expressive and concise programming language, typically set in an interactive environment. It was originally created, among other things, as a way to describe computers, by expressing mathematical notation in a rigorous way that could be interpreted by a computer. It is easy to learn but some APL programs can take some time to understand, especially for a newcomer. Few other programming languages offer the comprehensive array functionality of APL.

Unlike traditionally structured programming languages, code in APL is typically structured as chains of monadic or dyadic functions and operators acting on arrays. As APL has many nonstandard *primitives* (functions and operators, indicated by a single symbol or a combination of a few symbols), it does not have function or operator precedence. Early APL implementations did not have control structures (do or while loops, if-then-else), but by using array operations, usage of structured programming constructs was just not necessary. For example, the iota function (which yields a one-dimensional array, or vector, from 1 to N) can replace for-loop iteration. More recent implementations of APL generally include comprehensive control structures, so that data structure and program control flow can be clearly and cleanly separated.

The APL environment is called a **workspace**. In a workspace the user can define programs and data, i.e. the data values exist also outside the programs, and the user can manipulate the data without the necessity to define a program. For example,

$$N \leftarrow 4\ 5\ 6\ 7$$

assigns the vector values 4 5 6 7 to N;

$$N + 4$$

adds 4 to all values (giving 8 9 10 11) and prints them (a return value not assigned at the end of a statement to a variable using the assignment arrow ← is displayed by the APL interpreter);

$$+/N$$

prints the sum of N, i.e. 22.

The user can save the workspace with all values, programs and execution status.

APL is well-known for its use of a set of non-ASCII symbols which are an extension of traditional arithmetic and algebraic notation. Having single character names for SIMD vector functions is one way that APL enables compact formulation of algorithms for data transformation such as computing Conway's Game of Life in one line of code (example [38]). In nearly all versions of APL, it is theoretically possible to express any computable function in one expression, that is, in one line of code.

Because of its condensed nature and non-standard characters, APL has sometimes been termed a "write-only language", and reading an APL program can at first feel like decoding Egyptian hieroglyphics. Because of the unusual character set, many programmers use special keyboards with APL keytops for authoring APL code. Although there are various ways to write APL code using only ASCII characters,[39] in practice, it is almost never done. (This may be thought to support Iverson's thesis about notation as a tool of thought.)[40] Most if not all modern implementations use standard keyboard layouts, with special mappings or input method editors to access non-ASCII characters. Historically, the APL font has been distinctive, with uppercase italic alphabetic characters and upright numerals and symbols. Most vendors continue to display the APL character set in a custom font.

Advocates of APL claim that the examples of so-called write-only code are almost invariably examples of poor programming practice or novice mistakes, which can occur in any language. Advocates of APL also claim that they are far more productive with APL than with more conventional computer languages, and that working software can be implemented in far less time and with far fewer programmers than using other technology. APL lets an individual solve harder problems faster. Also, being compact and terse, APL lends itself well to larger scale software development as complexity arising from a large number of lines of code can be dramatically reduced. Many APL advocates and practitioners view programming in standard programming languages, such as COBOL and Java, as comparatively tedious. APL is often found where time-to-market is important, such as with trading systems.

Iverson later designed the J programming language which uses ASCII with digraphs instead of special symbols.

Execution

Interpreters

Today, most APL language activity takes place under the Microsoft Windows operating system, with some activity under Linux, Unix, and Mac OS. Comparatively little APL activity takes place today on mainframe computers.

APLNext (formerly APL2000) offers an advanced APL interpreter which operates under Linux, Unix, and Windows. It supports Windows automation, supports calls to operating system and user defined DLLs, has an advanced APL File System, and represents the current level of APL language development. APL2000's product is an advanced continuation of STSC's successful APL*Plus/PC and APL*Plus/386 product line.

Dyalog APL is an advanced APL interpreter which operates under Linux, Unix, and Windows. Dyalog has innovative extensions to the APL language which include new object oriented features, numerous language enhancements, plus a consistent namespace model used for both its Microsoft Automation interface, as well as native namespaces. For the Windows platform, Dyalog APL offers tight integration with Microsoft .Net, plus limited integration with the Microsoft Visual Studio development platform.

IBM offers a version of IBM APL2 for IBM AIX, Linux, Sun Solaris and Windows systems. This product is a continuation of APL2 offered for IBM mainframes. IBM APL2 was arguably the most influential APL system, which provided a solid implementation standard for the next set of extensions to the language, focusing on nested arrays.

NARS2000 is an open source APL interpreter written by Bob Smith, a well-known APL developer and implementor from STSC in the 1970s and 1980s. NARS2000 contains advanced features and new datatypes, runs natively under

Windows (32- and 64-bit versions), and runs under Linux and Apple Mac OS with Wine.

MicroAPL Limited offers *APLX*, a full-featured 64 bit interpreter for Linux, Windows, and Apple Mac OS systems. The core language is closely modelled on IBM's APL2 with various enhancements. *APLX* includes close integration with .NET, Java, Ruby and R.

Soliton Associates offers the SAX interpreter (Sharp APL for Unix) for Unix and Linux systems, which is a further development of I. P. Sharp Associates' Sharp APL product. Unlike most other APL interpreters, Kenneth E. Iverson had some influence in the way nested arrays were implemented in Sharp APL and SAX. Nearly all other APL implementations followed the course set by IBM with APL2, thus some important details in Sharp APL differ from other implementations.

Compilers

APL programs are normally interpreted and less often compiled. In reality, most APL compilers translated source APL to a lower level language such as C, leaving the machine-specific details to the lower level compiler. Compilation of APL programs was a frequently discussed topic in conferences. Although some of the newer enhancements to the APL language such as nested arrays have rendered the language increasingly difficult to compile, the idea of APL compilation is still under development today.

In the past, APL compilation was regarded as a means to achieve execution speed comparable to other mainstream languages, especially on mainframe computers. Several APL compilers achieved some levels of success, though comparatively little of the development effort spent on APL over the years went to perfecting compilation into machine code.

As is the case when moving APL programs from one vendor's APL interpreter to another, APL programs invariably will require changes to their content. Depending on the compiler, variable declarations might be needed, certain language features would need to be removed or avoided, or the APL programs would need to be cleaned up in some way. Some features of the language, such as the execute function (an expression evaluator) and the various reflection and introspection functions from APL, such as the ability to return a function's text or to materialize a new function from text, are simply not practical to implement in machine code compilation.

A commercial compiler was brought to market by STSC in the mid 1980s as an add-on to IBM's VSAPL Program Product. Unlike more modern APL compilers, this product produced machine code which would execute only in the interpreter environment, it was not possible to eliminate the interpreter component. The compiler could compile many scalar and vector operations to machine code, but it would rely on the APL interpreter's services to perform some more advanced functions, rather than attempt to compile them. However, dramatic speedups did occur, especially for heavily iterative APL code.

Around the same time, the book *An APL Compiler* by Timothy Budd appeared in print. This book detailed the construction of an APL translator, written in C, which performed certain optimizations such as loop fusion specific to the needs of an array language. The source language was APL-like in that a few rules of the APL language were changed or relaxed to permit more efficient compilation. The translator would emit C code which could then be compiled and run well outside of the APL workspace.

Today, execution speed is less critical and many popular languages are implemented using virtual machines - instructions that are interpreted at runtime. The Burroughs/Unisys **APLB** interpreter (1982) was the first to use dynamic incremental compilation to produce code for an APL-specific virtual machine. It recompiled on-the-fly as identifiers changed their functional meanings. In addition to removing parsing and some error checking from the main execution path, such compilation also streamlines the repeated entry and exit of user-defined functional operands. This avoids the stack setup and take-down for function calls made by APL's built-in operators such as Reduce and Each.

APEX, a research APL compiler, is available from Snake Island Research Inc. APEX compiles flat APL (a subset of ISO N8485) into SAC, a functional array language with parallel semantics, and currently runs under Linux. APEX-generated code uses loop fusion and array contraction, special-case algorithms not generally available to interpreters (e.g., upgrade of permutation vector), to achieve a level of performance comparable to that of Fortran.

The APLNext **VisualAPL** system is a departure from a conventional APL system in that VisualAPL is a true .Net language which is fully inter-operable with other .Microsoft .Net languages such as VB.Net and C#. VisualAPL is inherently object oriented and Unicode-based. While VisualAPL incorporates most of the features of standard APL implementations, the VisualAPL language extends standard APL to be .Net-compliant. VisualAPL is hosted in the standard Microsoft Visual Studio IDE and as such, invokes compilation in a manner identical to that of other .Net languages. By producing .Net common language runtime (CLR) code, it utilizes the Microsoft just-in-time compiler (JIT) to support 32-bit or 64-bit hardware. Substantial performance speed-ups over standard APL have been reported, especially when (optional) strong typing of function arguments is used.

An APL to C# translator is available from Causeway Graphical Systems. This product was designed to allow the APL code, translated to equivalent C#, to run completely outside of the APL environment. The Causeway compiler requires a run-time library of array functions. Some speedup, sometimes dramatic, is visible, but happens on account of the optimisations inherent in Microsoft's .Net framework.

A source of links to existing compilers is at APL2C [41].

Matrix optimizations

APL was unique in the speed with which it could perform complicated matrix operations. For example, a very large matrix multiplication would take only a few seconds on a machine which was much less powerful than those today. There were both technical and economic reasons for this advantage:

- Commercial interpreters delivered highly-tuned linear algebra library routines.
- Very low interpretive overhead was incurred per-array—not per-element.
- APL response time compared favorably to the runtimes of early optimizing compilers.
- IBM provided microcode assist for APL on a number of IBM/370 mainframes.

Phil Abrams' much-cited paper "An APL Machine" illustrated how APL could make effective use of lazy evaluation where calculations would not actually be performed until the results were needed and then only those calculations strictly required. An obvious (and easy to implement) lazy evaluation is the *J-vector* : when a monadic *iota* is encountered in the code, it is kept as a representation instead of being expanded in memory; in future operations, a *J-vector*'s contents are the loop's induction register, not reads from memory.

Although such techniques were not widely used by commercial interpreters, they exemplify the language's best survival mechanism: not specifying the order of scalar operations or the exact contents of memory. As standardized, in 1983 by ANSI working group X3J10, APL remains highly data-parallel. This gives language implementers immense freedom to schedule operations as efficiently as possible. As computer innovations such as cache memory, and SIMD execution became commercially available, APL programs ported with almost no extra effort spent re-optimizing low-level details.

Terminology

APL makes a clear distinction between *functions* and *operators*. Functions take arrays (variables or constants or expressions) as arguments, and return arrays as results. Operators (similar to higher-order functions) take functions or arrays as arguments, and derive related functions. For example the "sum" function is derived by applying the "reduction" operator to the "addition" function. Applying the same reduction operator to the "maximum" function (which returns the larger of two numbers) derives a function which returns the largest of a group (vector) of numbers. In the J language, Iverson substituted the terms 'verb' & ('adverb' or 'conjunction') for 'function' and

'operator'.

APL also identifies those features built into the language, and represented by a symbol, or a fixed combination of symbols, as *primitives*. Most primitives are either functions or operators. Coding APL is largely a process of writing non-primitive functions and (in some versions of APL) operators. However a few primitives are considered to be neither functions nor operators, most noticeably assignment.

Syntax

Examples

This displays "Hello, world":

```
'Hello, world'
```

This following immediate-mode expression generates a typical set of Pick 6 lottery numbers: six pseudo-random integers ranging from 1 to 40, *guaranteed non-repeating*, and displays them sorted in ascending order:

```
x[⍋x←6?40]
```

This combines the following APL functions:

- The first to be executed (APL executes from right to left) is the dyadic function "?" (named "Deal" when dyadic) that returns a vector consisting of a select number (left argument: 6 in this case) of random integers ranging from 1 to a specified maximum (right argument: 40 in this case), which, if said maximum ≥ vector length, is guaranteed to be non-repeating.
- This vector is then assigned to the variable x, because it is needed later.
- This vector is then sorted in ascending order by the monadic "⍋" function, which has as its right argument everything to the right of it up to the next unbalanced close-bracket or close-parenthesis. The result of ⍋ is the indices which will put its argument into ascending order.
- Then the output of ⍋ is applied to the variable x which we saved earlier, and it puts the items of x into ascending sequence.

Since there is no function to the left of the first x to tell APL what to do with the result, it simply outputs it to the display (on a single line, separated by spaces) without needing any explicit instruction to do that.

(Note that "?" also has a monadic equivalent called "Roll" which simply returns a single random integer between 1 and its sole operand [to the right of it], inclusive. Thus, a Role-Playing Game program might use the expression "?20" to roll a twenty-sided die.)

The equivalent of the above in a traditional verbose imperative/procedural programming language might follow an algorithm similar to this:

```
from random import shuffle # import randomization library
one_to_40 = range(1, 41) # generate list of numbers from 1 to 40
inclusive
shuffle(one_to_40) # randomly reorder the list
numbers = one_to_40[:6] # pick the first 6 elements from the list
numbers.sort() # sort the numbers in the list
print(" ".join(str(num) for num in numbers)) # convert numbers to
strings and output with spaces between them
```

Or using sample instead of shuffle and reduced at the cost of readability:

```
from random import sample
print(" ".join(str(n) for n in sorted(sample(range(1,41), 6))))
```

The following expression sorts a word list stored in matrix X according to word length:

```
X[⍋X+.≠' ';]
```

The following function "life", written in Dyalog APL, takes a boolean matrix and calculates the new generation according to Conway's Game of Life. It demonstrates the power of APL to implement a complex algorithm in very little code. But it is also very hard to follow unless one has an advanced knowledge of APL.

```
life←{↑1 ⍵∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂⍵}
```

In the following example, also Dyalog, the first line assigns some HTML code to a variable "txt" and then uses an APL expression to remove all the HTML tags, returning the text only as shown in the last line.

```
txt←'<html><body><p>This is <b>emphasized</b> text</body></html>'
⎕←{⍵/⍨~⍵∨≠\⍵}⍵∈'<>'}txt
This is emphasized text
```

The following expression finds all prime numbers from 1 to R. In both time and space, the calculation complexity is O(R²) (in Big O notation).

```
(~R∈R∘.×R)/R←1↓⍳R
```

Executed from right to left, this means:

- ⍳R creates a vector containing integers from 1 to R (if R = 6 at the beginning of the program, ⍳R is 1 2 3 4 5 6)
- Drop first element of this vector (↓ function), i.e. 1. So 1↓⍳R is 2 3 4 5 6
- Set R to the new vector (←, assignment primitive), i.e. 2 3 4 5 6
- Generate outer product of R multiplied by R, i.e. a matrix which is the *multiplication table* of R by R (∘.× function), i.e.

4	6	8	10	12
6	9	12	15	18
8	12	16	20	24
10	15	20	25	30
12	18	24	30	36

- Build a vector the same length as R with 1 in each place where the corresponding number in R is in the outer product matrix (∈, set inclusion function), i.e. 0 0 1 0 1
- Logically negate the values in the vector (change zeros to ones and ones to zeros) (~, negation function), i.e. 1 1 0 1 0
- Select the items in R for which the corresponding element is 1 (/ function), i.e. 2 3 5

(Note, this assumes the APL origin is 1, i.e., indices start with 1. APL can be set to use 0 as the origin, which is convenient for some calculations.)

Character set

APL has always been criticized for its choice of a unique, non-standard character set. The observation that some who learn it usually become ardent adherents shows that there is some weight behind Iverson's idea that the notation used does make a difference. In the beginning, there were few terminal devices which could reproduce the APL character set—the most popular ones employing the IBM Selectric print mechanism along with a special APL type element. Over time, with the universal use of high-quality graphic display, printing devices and Unicode support, the APL character font problem has largely been eliminated; however, the problem of entering APL characters requires the use of input method editors or special keyboard mappings, which may frustrate beginners accustomed to other programming languages.

Usage

APL has long had a small and fervent user base. It was and still is popular in financial and insurance applications, in simulations, and in mathematical applications. APL has been used in a wide variety of contexts and for many and varied purposes. A newsletter titled "Quote-Quad" dedicated to APL has been published since the 1970s by the SIGAPL section of the Association for Computing Machinery (Quote-Quad is the name of the APL character used for text input and output).

Before the advent of full-screen systems and until as late as the mid-1980s, systems were written such that the user entered instructions in his own business specific vocabulary. APL timesharing vendors delivered applications in this form. On the I. P. Sharp timesharing system, a workspace called 39 MAGIC offered access to financial and airline data plus sophisticated (for the time) graphing and reporting. Another example is the GRAPHPAK workspace supplied with IBM's APL2.

Because of its matrix operations, APL was for some time quite popular for computer graphics programming, where graphic transformations could be encoded as matrix multiplications. One of the first commercial computer graphics houses, Digital Effects, based in New York City, produced an APL graphics product known as "Visions," which was used to create television commercials and film animation for the 1982 film Tron (Digital Effects use of APL was informally described at a number of SIGAPL conferences in the late 1980s, examples discussed included the early UK Channel 4 TV logo/ident. What is not clear is the extent to which APL was directly involved in the making of "Tron", and at this point in time the reference is more of an urban legend or historic curio than much else).

Interest in APL has steadily declined since the mid 1980s. This was partially due to the lack of a smooth migration path from higher performing mainframe implementations to low-cost personal computer alternatives, as APL implementations for computers before the Intel 80386 released in the late 1980s were only suitable for small applications. The growth of end-user computing tools such as Microsoft Excel and Microsoft Access also eroded into potential APL usage. These are appropriate platforms for what may have been mainframe APL applications in the 1970s and 1980s. Some APL users migrated to the J programming language, which offers more advanced features. Lastly, the decline was also due in part to the growth of MATLAB, GNU Octave, and Scilab. These scientific computing array-oriented platforms provide an interactive computing experience similar to APL, but more resemble conventional programming languages such as Fortran, and use standard ASCII.

Notwithstanding this decline, APL finds continued use in certain fields, such as accounting research.[42]

Standardization

APL has been standardized by the ANSI working group X3J10 and ISO/IEC Joint Technical Committee 1 Subcommittee 22 Working Group 3. The Core APL language is specified in ISO 8485:1989, and the Extended APL language is specified in ISO/IEC 13751:2001.

Glossary

Some words used in APL literature have meanings that differ from those in both mathematics and the generality of computer science.

term	description
function	operation or mapping that takes zero, one (right) or two (left & right) array valued arguments and may return an array valued result. A function may be:
	Primitive - built-in and represented by a single glyph;[43]
	Defined - as a named and ordered collection of program statements;[43]
	Derived - as a combination of an operator with its arguments.[43]
array	data valued object of zero or more orthogonal dimensions in row-major order in which each item is a primitive scalar datum **or another array**.[44]
niladic	not taking or requiring any arguments,[45]
monadic	requiring only one argument; on the right for a function, on the left for an operator, unary[45]
dyadic	requiring both a left and a right argument, binary[45]
ambivalent or nomadic	capable of use in a monadic or dyadic context, permitting its left argument to be elided[43]
operator	operation or mapping that takes one (left) or two (left & right) function or array valued arguments (operands) and derives a function. An operator may be:
	Primitive - built-in and represented by a single glyph;[43]
	Defined - as a named and ordered collection of program statements.[43]

Criticism

APL has been used since the mid 1960s on mainframe computers and has itself evolved in step with computers and the computing market. APL is not widely used, but minimalistic and high-level by design, at several points in its history it could have captured a more significant market share, but never did. APL first appeared on large mainframe computers and, like C and Pascal, did find its way to CP/M and MS-DOS based microcomputers. Unlike C and Pascal, APL did not flourish in the microcomputer arena until hardware of sufficient computing power became commonly available. Perhaps accounting for its lack of mainstream appeal, APL's characteristics have always led to much criticism of the language. As always, such complaints may arise from misconceptions, have origins in distant APL history and no longer be relevant today, or they may have some degree of validity. This article is concerned only with the latter two categories.

Footnotes

[1] "A Bibliography of APL and J" (http://www.jsoftware.com/jwiki/Essays/Bibliography). Jsoftware.com. . Retrieved 2010-02-03.

[2] "Kx Systems - An Interview with Arthur Whitney - Jan 2004" (http://kx.com/Company/press-releases/arthur-interview.php). Kx.com. 2004-01-04. . Retrieved 2010-02-03.

[3] "The Growth of MatLab - Cleve Moler" (http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf) (PDF). . Retrieved 2010-02-03.

[4] "About Q'Nial" (http://www.nial.com/AboutQNial.html). Nial.com. . Retrieved 2010-02-03.

[5] Iverson, Kenneth E. (1962). *A Programming Language* (http://www.softwarepreservation.org/projects/apl/book/APROGRAMMING LANGUAGE/view). Wiley. ISBN 0471430145. .

[6] "an experimental APL interpreter" (http://www.nars2000.org/). NARS2000. . Retrieved 2010-02-03.

[7] "Dyalog V12 Platforms" (http://www.dyalog.com/version12.html). Dyalog.com. . Retrieved 2010-02-03.

[8] "The Future of APL in the Insurance World. Gary A. Bergquist" (http://portal.acm.org/ft_gateway.cfm?id=347203&type=pdf&coll=GUIDE&dl=GUIDE&CFID=46163917&CFTOKEN=91975100). Portal.acm.org. . Retrieved 2010-02-03.

[9] "APLX version 4 – from the viewpoint of an experimental physicist. Vector 23.3" (http://www.vector.org.uk/archive/v233/webber.htm). Vector.org.uk. 2008-05-20. . Retrieved 2010-02-03.

[10] OOSTATS – A New Approach to Statistics via APL (http://conference.dyalog.com/2008Elsinore/Presentations/Alan Sykes - Dyalog08.pdf)

[11] "ACM Award Citation – John Backus. 1977" (http://awards.acm.org/citation.cfm?id=0703524&srt=all&aw=140&ao=AMTURING). Awards.acm.org. 1924-12-03. . Retrieved 2010-02-03.

[12] "Agile Approach" (http://www.agileapproach.co.uk/). Agile Approach. . Retrieved 2010-02-03.

[13] Iverson, Kenneth E., "Automatic Data Processing: Chapter 6: A programming language" (http://www.softwarepreservation.org/projects/apl/book/Iverson-AutomaticDataProcessing-color.pdf/view), 1960, DRAFT copy for Brooks and Iverson 1963 book, "Automatic Data Processing".

[14] Brooks, Fred; Iverson, Kenneth, (1963), *Automatic Data Processing*, John Wiley & Sons Inc.

[15] Hellerman, H., "Experimental Personalized Array Translator System", *Communications of the ACM*, 7, 433 (July, 1964).

[16] Falkoff, Adin D.; Iverson, Kenneth E., "The Evolution of APL" (http://www.jsoftware.com/papers/APLEvol.htm), ACM SIGPLAN Notices 13, 1978-08.

[17] Abrams, Philip S., *An interpreter for "Iverson notation"* (http://infolab.stanford.edu/TR/CS-TR-66-47.html), Technical Report: CS-TR-66-47, Department of Computer Science, Stanford University, August 1966.

[18] Haigh, Thomas, "Biographies: Kenneth E. Iverson", *IEEE Annals of the History of Computing*, 2005

[19] Breed, Larry, "The First APL Terminal Session" (http://portal.acm.org/citation.cfm?id=138094.140933), *APL Quote Quad*, Association for Computing Machinery, Volume 22, Number 1, September 1991, p.2-4.

[20] Adin Falkoff (http://www.computerhistory.org/tdih/?setdate=19/12/2009) - Computer History Museum. "Iverson credited him for choosing the name APL and the introduction of the IBM golf-ball typewriter with the replacement typehead, which provided the famous character set to represent programs."

[21] Larry Breed (August 2006). "How We Got to APL\1130" (http://www.vector.org.uk/archive/v223/APL_1130.htm). *Vector (British APL Association)* **22** (3). ISSN 0955-1433. .

[22] APL\1130 Manual (http://bitsavers.org/pdf/ibm/1130/lang/1130-03.3.001_APL_1130_May69.pdf), May 1969

[23] Falkoff, Adin; Iverson, Kenneth E., "APL\360 Users Guide" (http://bitsavers.org/pdf/ibm/apl/APL_360_Users_Manual_Aug68.pdf), IBM Research, Thomas J. Watson Research Center, Yorktown Heights, NY, August 1968.

[24] "APL\360 Terminal System" (http://bitsavers.org/pdf/ibm/apl/APL_360_Terminal_System_Mar67.pdf), IBM Research, Thomas J. Watson Research Center, March 1967.

[25] Pakin, Sandra (1968). *APL\360 Reference Manual*. Science Research Associates, Inc.. ISBN 0-574-16135-X.

[26] Falkoff, Adin D.; Iverson, Kenneth E., *The Design of APL* (http://www.research.ibm.com/journal/rd/174/ibmrd1704F.pdf), *IBM Journal of Research and Development*, Volume 17, Number 4, July 1973. "These environmental defined functions were based on the use of still another class of functions — called "I-beams" because of the shape of the symbol used for them — which provide a more general facility for communication between APL programs and the less abstract parts of the system. The I-beam functions were first introduced by the system programmers to allow them to execute System/360 instructions from within APL programs, and thus use APL as a direct aid in their programming activity. The obvious convenience of functions of this kind, which appeared to be part of the language, led to the introduction of the monadic I-beam function for direct use by anyone. Various arguments to this function yielded information about the environment such as available space and time of day."

[27] "Turing Award Citation 1979" (http://awards.acm.org/citation.cfm?id=9147499&srt=all&aw=140&ao=AMTURING). Awards.acm.org. . Retrieved 2010-02-03.

[28] Falkoff, Adin D. (1991). "The IBM family of APL systems" (http://google.com/search?q=cache:WGQh6w1-YHAJ:www.research.ibm.com/journal/sj/304/ibmsj3004C.pdf) (PDF). *IBM Systems Journal* (IBM) **30** (4): 416–432. doi:10.1147/sj.304.0416. Archived from the original (http://www.research.ibm.com/journal/sj/304/ibmsj3004C.pdf) on (unknown). . Retrieved 2009-06-13.

[29] "VideoBrain Family Computer" (http://books.google.com/books?id=OQEAAAAAMBAJ&pg=PA133&lpg=PA133&dq=videobrain+family+computer+apl/s&source=bl&ots=_tmStYA0UG&sig=mxb5bqgWuA_NBVww1ywhpA1iNWY&hl=en&

ei=rleIS8_hPN2mtgez8vi0DQ&sa=X&oi=book_result&ct=result&resnum=5&ved=0CBQQ6AEwBA#v=onepage&q=videobrain family computer apl/s&f=false), *Popular Science*, November 1978, advertisement.

[30] Higgins, Donald S., "PC/370 virtual machine" (http://portal.acm.org/citation.cfm?id=382167.383025), *ACM SIGSMALL/PC Notes*, Volume 11, Issue 3 (August 1985), pp.23 - 28, 1985.

[31] "APLX : New features in Version 5" (http://www.microapl.co.uk/apl/aplxv5.html). Microapl.co.uk. 2009-04-01. . Retrieved 2010-02-03.

[32] "APL2000 NetAccess" (http://www.apl2000.com/netaccess.php). Apl2000.com. . Retrieved 2010-02-03.

[33] "Introduction to Visual APL" (http://www.visualapl.com/library/aplnext/Visual_APLNext.htm). Visualapl.com. . Retrieved 2010-02-03.

[34] "Dyalog Built-in Charting" (http://www.dyalog.com/version12.html). Dyalog.com. . Retrieved 2010-02-03.

[35] "XML-to-APL Conversion tool in Dyalog APL and APLX" (http://www.dyalog.com/help/12.1/html/xml convert.htm). Dyalog.com. . Retrieved 2010-02-03.

[36] http://foldoc.org/lambda+expression

[37] "Dynamic Functions in Dyalog APL - John Scholes. 1997" (http://www.dyalog.com/download/dfns.pdf) (PDF). . Retrieved 2010-02-03.

[38] http://catpad.net/michael/apl

[39] Dickey, Lee, A list of APL Transliteration Schemes (http://www.math.uwaterloo.ca/apl_archives/apl/translit.schemes), 1993

[40] Iverson K.E.," Notation as a tool of thought (http://elliscave.com/APL_J/tool.pdf)", *Communications of the ACM*, 23: 444-465 (August 1980).

[41] http://www.apl2c.de/home/Links/links.html

[42] "Stanford Accounting PhD requirements" (http://www.gsb.stanford.edu/phd/fields/accounting/index.html). Gsb.stanford.edu. . Retrieved 2010-02-03.

[43] "APL concepts" (http://www.microapl.co.uk/APL/apl_concepts_chapter6.html). Microapl.co.uk. . Retrieved 2010-02-03.

[44] "Nested array theory" (http://www.nial.com/ArrayTheory.html). Nial.com. . Retrieved 2010-02-03.

[45] "Programmera i APL", Bohman, Fröberg, Studentlitteratur, ISBN-91-44-13162-3

References

- Falkoff, Adin D.; Iverson, Kenneth E.; Sussenguth, Edward H. (1964). "A Formal Description of SYSTEM/360" (http://web.archive.org/web/20080227012111/http://www.research.ibm.com/journal/sj/032/falkoff.pdf). *IBM Systems Journal* (New York) **3** (3). Archived from the original (http://www.research.ibm.com/journal/sj/032/falkoff.pdf) on February 27, 2008.
- *History of Programming Languages*, chapter 14
- Banon, Gerald Jean Francis (1989). *Bases da Computacao Grafica*. Rio de Janeiro: Campus. p. 141.
- LePage, Wilbur R. (1978). *Applied A.P.L. Programming*. Prentice Hall.

Further reading

- *An APL Machine* (http://www.slac.stanford.edu/pubs/slacreports/slac-r-114.html) (1970 Stanford doctoral dissertation by Philip Abrams)
- *A Personal History Of APL* (http://ed-thelen.org/comp-hist/APL-hist.html) (1982 article by Michael S. Montalbano)
- McIntyre, Donald B. (1991). "Language as an intellectual tool: From hieroglyphics to APL" (http://web.archive.org/web/20060504050437/http://www.research.ibm.com/journal/sj/304/ibmsj3004N.pdf). *IBM Systems Journal* **30** (4). Archived from the original (http://www.research.ibm.com/journal/sj/304/ibmsj3004N.pdf) on May 4, 2006.
- Iverson, Kenneth E. (1991). "A Personal view of APL" (http://web.archive.org/web/20080227012149/http://www.research.ibm.com/journal/sj/304/ibmsj3004O.pdf). *IBM Systems Journal* **30** (4). Archived from the original (http://www.research.ibm.com/journal/sj/304/ibmsj3004O.pdf) on February 27, 2008.
- *A Programming Language* (http://www.softwarepreservation.org/projects/apl/book/APROGRAMMING LANGUAGE/view) by Kenneth E. Iverson
- Brooks, Frederick P.; Kenneth Iverson (1965). *Automatic Data Processing, System/360 Edition*. ISBN 0-471-10605-4.

External links

- APL Wiki (http://aplwiki.com/)
- NARS2000 Open Source APL (http://www.nars2000.org/)
- comp.lang.apl (http://groups.google.com/group/comp.lang.apl/topics)
- IBM APL2 (http://www-306.ibm.com/software/awdtools/apl/)
- Dyalog APL (http://www.dyalog.com/)
- APL2000 (http://www.apl2000.com/)
- APLNext: APL for .Net (http://www.aplnext.com/)
- MicroAPL Ltd. (http://www.microapl.co.uk/apl/index.html)
- SIGAPL Home Page (http://www.sigapl.org/)
- Quote-Quad newsletter (http://www.sigapl.org/qq.htm)
- British APL Association's journal *Vector* (http://www.vector.org.uk/)
- APL-Journal - German APL-Publication (http://www.rhombos.de/shop/c/categorie/APLJournal-00021/Uebersicht-00134/)
- OOPAL: Integrating Array Programming in Object-Oriented Programming (http://www.fscript.org/documentation/OOPAL.pdf)
- An introduction to Object Oriented APL (http://web.archive.org/web/20080228023208/http://www.dyalog.dk/whatsnew/OO4APLERS.pdf)
- Comparison of Black-Scholes options pricing model in many languages, including APL (http://www.espenhaug.com/black_scholes.html)
- System Building with APL + Win (http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470030208.html)
- Why APL? (http://www.sigapl.org/whyapl.htm)
- About APL by Brad McCormick (http://www.users.cloud9.net/~bradmcc/APL.html)
- by Rex Swain (http://www.rexswain.com/aplinfo.html)
- by Eric Lescasse (http://www.lescasse.com/)
- Sam Sirlin's APL FAQ (http://home.earthlink.net/~swsirlin/apl.faq.html) (Frequently Asked Questions list) plus link to versions of Budd's compiler
- Java applet to practice APL (http://lparis45.free.fr/apl.html)
- APL Unicode Font – Extended (http://www.vector.org.uk/resource/simp2.htm), a free font containing all the Unicode glyphs for the APL function symbols.
- IBM 3270 keyboard layout for APL (http://publib.boulder.ibm.com/infocenter/pcomhelp/v5r9/topic/com.ibm.pcomm.doc/reference/html/kbd_reference05.htm#FIGTYPE1)

Article Sources and Contributors

APL (programming language) *Source*: http://en.wikipedia.org/w/index.php?oldid=412776156 *Contributors*: 212.186.10.xxx, 5jt, A2Kafir, ABCD, Aaron Rotenberg, Agateller, Ais523, Alastair Haines, Alexander256, AliasMarlowe, Andonic, Andthepharaohs, Angr, Anomalocaris, AnonMoos, ApMog, Archanamiya, Arichnad, B jonas, Bachcell, Balrog-kun, Baltar, Gaius, Barticus88, BenFrantzDale, Billgordon1099, BudgieJane, C. A. Russell, CRGreathouse, CS46, Catpad, Chaos5023, ChrisCork, Christian List, Cnilep, Cohesion, Colonies Chris, Conversion script, Cowplopmorris, Cowznofski, Craig Stuntz, Cuye, Cybercobra, DESiegel, DNewhall, Dachshund, Damian Yerrick, Danakil, DavidDouthitt, Dcoetzee, DevSolar, Dick Bowman, Dinsdale Doug Piranha, Dmh, DouglasGreen, Dougmerritt, DragonflySixtyseven, Dysprosia, EdC, Edward, Ehrenkater, Empro2, Erydo, Etm3, Everyking, Faradayplank, Fatsamsgrandslam, Fester Q Bestertester, Folajimi, Frap, Frecklefoot, Freital, Frungi, Furchild, GJaxon, Gauge, Geeperzcreeperz, Geira, GeorgHH, Giftlite, Gimboid13, Gingerjoos, Glennklockwood, Gnomon, Greenrd, Grendelkhan, Gro-Tsen, H, Hairy Dude, Hajhouse, Hongooi, Imz, Inter, Itai, JJL, JLaTondre, Jack Merridew, Jacob Myers, JensAlfke, JerroldPease-Atlanta, Jerryobject, JidGom, Jim62sch, Jleedev, Jmath666, Johanvranken, John Vandenberg, Jon186, Jshadias, Jsnx, Julesd, KSmrq, Kaithomasmax, Kallikanzarid, KeithWaclena, Kleg, KymFarnik, Lambiam, Lee Daniel Crocker, Leibniz, Lightmouse, Lparis45, MIT Trekkie, Macrakis, Magnus.de, Mav, Metamorf, Mezzaluna, Michaeltarpley, Monadic Mike, Monedula, Mumphisman, NSH001, NVI, NevilleDNZ, Nick Garvey, Nikai, Nikola Smolenski, Norm mit, Ntsimp, Od Mishehu, OlEnglish, Parsiferon, PeterJeremy, Phe, Phil Boswell, Phil Last, Piet Delport, Pmcjones, Poor Yorick, Postagoras, Profoss, Psd, Quuxplusone, Qwfp, R'n'B, R3m0t, RaulMiller, RayTrimble, Raysonho, RedWolf, Reedbeta, Reedy, Reeve, Renaldo Moon, Renku, Rfc1394, Rjwilmsi, Rlw, RobyWayne, Roger Hui, RossPatterson, Rursus, Ruud Koot, SGBailey, Sannse, Sen Mon, Sensiworld, Shadowjams, Shoaler, Shot, Sigmundur, Silviubogan, SimDarthMaul, Simicich, Skipcave, Soulpatch, Sperling, Ssd, Stan Shebs, Stevertigo, Studio17, Szumyk, Taejo, Tcp-ip, Teledon, Template namespace initialisation script, The Mighty Clod, The doctor23, TheAllSeeingEye, Thumperward, Timothy Campbell, TomTrottier, Torc2, Trey314159, Tuntable, UTF-8, Uriber, UtherSRG, Vina, Vnyx, Wellington, Welsh, Who, Wikiklrsc, Wlthomassen, Ww, Wws, Zaxby, ZeroOne, Zoicon5, Δ, 212 anonymous edits

Image Sources, Licenses and Contributors

Image:IBM Selectric Globe Wiki.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:IBM_Selectric_Globe_Wiki.jpg *License*: Public Domain *Contributors*: de:hd

Image:APL-keybd2.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:APL-keybd2.svg *License*: GNU Free Documentation License *Contributors*: User:Rursus

Image:LifeInApl.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:LifeInApl.gif *License*: Public Domain *Contributors*: Kaithomasmax

Image:StripHtmlFromText.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:StripHtmlFromText.gif *License*: Public Domain *Contributors*: Kaithomasmax

License