An Introduction to Array Thinking

Before I introduce "array thinking" by way of an example of what it is and what it isn't, I'd like to give some background and provide context about why this is important. The following blog entry from Scott Locklin sets the tone.

Ruins of forgotten empires: APL languages

Posted in **Design**, **J**, **Lush** by Scott Locklin on July 28, 2013

One of the problems with modern computer technology: programmers don't learn from the great masters. There is such a thing as a Beethoven or Mozart of software design. Modern programmers seem more familiar with Lady Gaga. It's not just a matter of taste and an appreciation for genius. It's a matter of forgetting important things.

Talk to the hand that made APL.

[From http://scottlocklin.wordpress.com/2013/07/28/ruins-of-forgotten-empires-apl-languages/]

If you have followed the venerable news group "Risks in Computing" (see http://catless.ncl.ac.uk/Risks/), one of the recurring themes is how many well-known risks are commonly ignored in software projects. It's as if the computing profession deliberately ignores the lessons of the past, or it learns the wrong ones. I'm reminded of this summary of the works of early nineteenth-century psychologist Lev Vygotsky talking about "tools of the mind":

"According to Vygotsky, until children learn to use mental tools, their learning is largely controlled by the environment; they attend only to the things that are brightest or loudest, and they can remember something only if has been repeated many times."

[From http://www.toolsofthemind.org/philosophy/vygotskian-approach/]

Vigotsky goes on to say that after "...children master mental tools, they become in charge of their own learning, by attending and remembering in an intentional and purposeful way."

Wrong Lessons from History

But it's even worse than forgetting the valuable insights of past "giants": as a field, computing often seems to seize hold of mistaken notions, or at least ones poorly supported by empirical evidence. Here's an example pertinent to the larger issue of array thinking we hope to address here.

Truth, APL and the Dijkstra problem

Published on October 12, 2010 in APL/J/K The Movie, CS Roots and History. 11 Comments byaprogramminglanguage

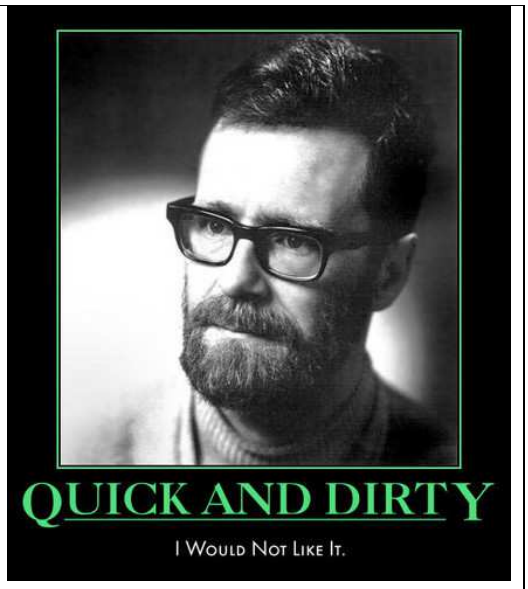
Never underestimate the power of a tag line!

In June of 1975, Edsger W. Dijkstra wrote an essay called: *How do we tell truths that might hurt?* Which is characterized as a series of aphorisms about computer programming languages, one of which is APL.

Essentially, Dijkstra wrote a bunch of catchy, satirical critiques about the programming languages of the day. He prefaces this work by mentioning a lack of rigorous criticism in the computing community, which I haven't experienced, but I believe, because it's backed up by some of Dennis Shasha's work. So, I suspect that as an artifact *of its time*, this work could have galvanized computer scientists to shape up.

And then time marches on…

I haven't done an official count, but my initial investigations indicate that Dijkstra's*1975* quip against APL is the most frequently used quote about APL in cyberspace *to this day*.



Edsgar Dijkstra

[From http://lathwellproductions.ca/wordpress/2010/10/12/truth-apl-and-the-dijkstra-problem/]

So, what can be done about this? For one thing, we can keep bringing up these good old ideas and emphasizing them with concrete examples to clarify why they are good ideas.

One such idea is the general notion of "array thinking". This is an array-oriented way of working on computational problems perhaps best illustrated by an example.

Simplifying Code with Array Thinking

One of the newer members of the J Forum, Joe Bogner, posted an example of some code he was attempting to re-write in J. The purpose of this code, which he had written in C#, was to simply parse a large body of code to measure a few aspects of its complexity. Specifically, in this case, the code is generating statistics about the nesting levels of parenthesized expressions in an arbitrary piece of (Java) code.

The core of the example code looked something like this:

```
for(int i = 0; i < text.Length; i++) {
        var currentChar = text[i];
        if (i < text.Length-1 && text[i] == '@' && text[(i+1)] == '{') {
                state = CODEBLOCK;
                depth = 0;
                longestBlock = chars.Count > longestBlock ? chars.Count : longestBlock;
                chars.Clear();
                blocks += 1;
        }
```

```
        if (new string(text.Skip(i).Take(SCRIPT_START.Length).ToArray()).ToLower() ==
SCRIPT_START) {
                state = SCRIPT;
                chars.Clear();
                i += SCRIPT_START.Length;
        }

        if (state == CODEBLOCK) {
                chars.Add(currentChar);
                if (currentChar == '{') {
                        depth += 1;
                        maxDepth = depth > maxDepth ? depth - 1: maxDepth;
                }
                if (currentChar == '}') {
                        depth -= 1;
                        if (depth == 0) {
                                state = NONE;
                        }
                }
        }
        if (state == SCRIPT) {
                if (new string(text.Skip(i).Take(SCRIPT_END.Length).ToArray()).ToLower()
== SCRIPT_END) {
                        state = NONE;
                        scripts += 1;
                        i += SCRIPT_END.Length;
                        var scriptBlock = new String(chars.Skip(1).ToArray());
                        longestScript = scriptBlock.Length > longestScript ?
scriptBlock.Length : longestScript;
                }
                chars.Add(currentChar);
        }
}
```

The result of the above code would be statistics as shown here:

```
log("Max Depth", maxDepth);
log("# of blocks", blocks);
log("Longest block", longestBlock);
log("Scripts", scripts);
log("Longest script", longestScript);
```

The members of the J Forum offered extensive comments about how to render this code in idiomatic J, with a lengthy discussion about a “state machine” solution and how this might be achieved. This discussion clearly took the code sample as its starting point, so was greatly influenced by the existing, non-array-oriented, solution.

I had not followed the discussion very closely, but, after it had been going on for a while, I thought of something to contribute to solving the problem. However, I almost did not even post this idea as it was based on something very simple I’d learned years before when working in APL and I was under the mistaken idea that it must be widely known.

When I did post the sketch of a solution, as follows, the response from the original problem-submitter was dramatic: he was astonished and deeply gratified by the insight of this simple, array-oriented, idea. My

contribution was simply this example code fragment applied to a simple example of a short string with a few levels of parenthesis nesting:

This (old) trick works nicely if you can fit complete expressions into memory:

```
   +/\(1 _1 0){~'{}' i. '@{ foo; if (abc) { if (q) { m; } } }'
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 2 2 1 1 0
```

This result is a vector of numbers of the nesting level corresponding to each character in the input string.

Old Tricks for New Dogs

I was alerted to just how old this trick is by a note from Roger Hui who referred me to something written by Alan Perlis in the early 1960's in which he recounts his experience at a meeting back then.

```
from:  Roger Hui <rogerhui.canada@gmail.com>
date:  Wed, Jan 15, 2014 at 11:22 AM
```

An oldie but a goodie. See the third paragraph of Perlis, APL is more French than English <http://www.jsoftware.com/papers/perlis78.htm>, 1978, talking about events in the 1960s.

The relevant excerpt from that paper is as follows:

> I was at a meeting in Newcastle, England, where I'd been invited to give a talk, as had Don Knuth of Stanford, Ken Iverson from IBM, and a few others as well. I was sitting in the audience sandwiched between two very esteemed people in computer science and computing — Fritz Bauer, who runs computing in Bavaria from his headquarters in Munich, and Edsger Dijkstra, who runs computing all over the world from his headquarters in Holland.
>
> Ken was showing some slides — and one of his slides had something on it that I was later to learn was an APL one-liner. And he tossed this off as an example of the expressiveness of the APL notation. I believe the one-liner was one of the standard ones for indicating the nesting level of the parentheses in an algebraic expression. But the one-liner was very short — ten characters, something like that — and having been involved with programming things like that for a long time and realizing that it took a reasonable amount of code to do, I looked at it and said, "My God, there must be something in this language."

The simple solution I'd offered to the recent problem is clearly a descendant of this "ten characters" written on a chalkboard about fifty years ago. Not only that, but Perlis's epiphany is closely echoed decades later:

```
from:  Joe Bogner joebogner@gmail.com
date:  Wed, Jan 15, 2014 at 12:47 PM
```

I looked at Devon's answer and had a response very similar to the paper (before reading the paper). It'd be like if I was riding a horse and carriage in 1910 and an automobile raced by. J has a pleasant way of humbling in 10 characters or less.

Another Kind of Reaction

Interestingly enough, this same account from Perlis continues with the following, very different reaction from some of the influential members of the audience with whom he was sitting:

> ... I looked at it and said, "My God, there must be something in this language." Bauer, on my left, didn't see that. What he saw or heard was Ken's remark that APL is an extremely appropriate language for teaching algebra, and he muttered under his breath to me, in words I will never

> forget, “As long as I am alive, APL will never be used in Munich.” And Dijkstra, who was sitting on my other side, leaned toward Bauer and said, “Nor in Holland.” The three of us were listening to the same lecture, but we obviously heard different things.

Here we see the hostility, to which we alluded earlier, of Dijkstra toward APL. This hostility seems oddly out of character based on many of his other writings but, regardless, it seems to have echoed through the history of computer programming. In any case, let the rehabilitation begin.

How does Array Thinking Help Us?

One way in which array thinking seems to simplify a problem is by allowing us to think in terms of determinate explicit data structures rather than in terms of procedural processing. Whereas the procedural approach forces us to keep numerous, small things in mind, as exemplified by lines like the following, an array-oriented approach allows us to deal with fewer items.

```
if (currentChar == '{') {
        depth += 1;
        maxDepth = depth > maxDepth ? depth - 1: maxDepth;
}
if (currentChar == '}') {
        depth -= 1;
        if (depth == 0) {
                state = NONE;
        }
}
```

As opposed to something like this intermediate result:

```
   +/\(1 _1 0){~'{}' i. '@{ foo; if (abc) { if (q) { m; } } }'
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 2 2 1 1 0
```

This latter example shows us a vector of the nesting levels of each character in the input string. Unlike the cognitive load imposed by the procedural code, this sort of display maintains relevant information on the page in front of our eyes, reducing the number of things we must keep active in our short-term memory. Not only that, but it provides a simple sanity check by allowing us to compare this intermediate result to the original string visually. We could do this even more precisely like this:

```
   string=. '@{ foo; if (abc) { if (q) { m; } } }'  NB. Test string
   countNesting=: 13 : '+/\(1 _1 0){~x i. y'        NB. Basic nesting-level count
   nestLvl=. '{}' countNesting string               NB. Example result
   string,:' '-.~":nestLvl                          NB. Result aligned with input
@{ foo; if (abc) { if (q) { m; } } }
01111111111111111222222223333322110
```

Seeing this gives us confidence that we’re on the right track of a solution. Not only that, by abstracting out the array of the parenthesis pair (`'{}'`) as the left argument, we’ve built a generalizable, re-usable piece of code instead of hard-coding the type of parentheses into the code as the procedural version does.

Continuing Education

Since presenting an introduction like this at the J Conference and at Iverson College, I’ve been soliciting good examples from the array-programming community. I hope to put these together as part of a larger work to explain this useful, old idea.

Please feel free to send any examples to devon@acm.org. Thank you.