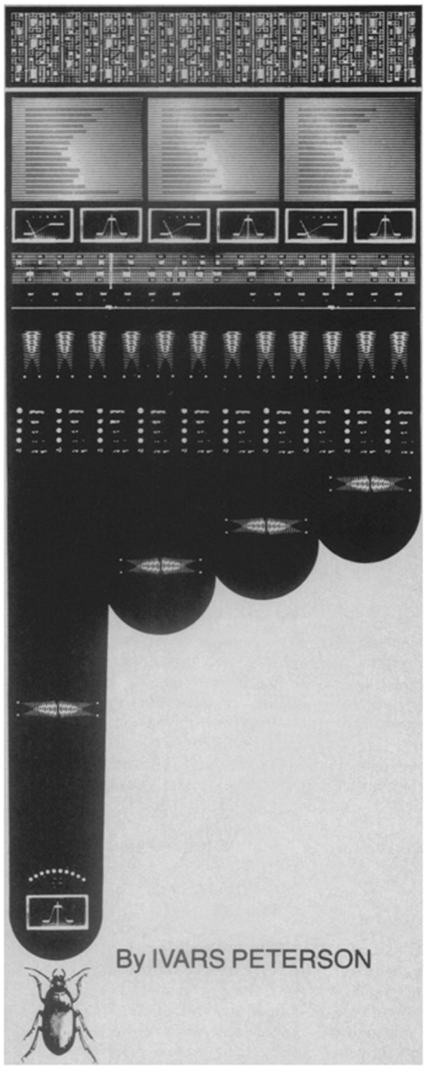

John Pack

COMPUTER LANGUAGES: IN SEARCH OF A BETTER BUG FINDER

Two contrasting approaches to computer language development may someday help beginners avoid some of the frustrations of programming a computer

My personal computer is blinking at me — again. The video screen's winking cursor, steadily flashing on and off, is almost hypnotic. For several hours, it has patiently pointed out error after error in the computer program that I am trying to write. This time, the problem is in line 3890. A colon sits where a semicolon ought to go.

Not all my mistakes are this easy to find or correct. Some are the consequences of hitherto hidden quirks built into the particular computer language, BASIC, that I am using. These quirks often catch me by surprise, and programming around them takes a little bit of ingenuity and a lot of time.

I also know that even when the computer's error-sniffing nose finally locates all of my "syntax errors," and the computer program actually runs, the program may still do something that I did not anticipate. Somewhere in the excruciatingly detailed path I had to set for the computer's digital manipulations may lie unintended branches or loops that take the program far afield.

The frustrations of writing a computer program are familiar to both novices like me and experts like John Backus of the IBM Research Laboratory in San Jose, Calif. He says, "I've sat in front of my own personal computer and cursed the designers of BASIC . . . just as you have."

Backus was part of the IBM team that in the 1950s developed FORTRAN, the first widely used programming language. It allowed programmers to escape the tedious task of writing out instructions as strings of 1s and 0s. Almost all current computer languages, including BASIC, are descendants of this first effort.

The list of programming languages available grows year by year, much to the consternation of computer users who are faced with difficult choices over which language is best for their purpose. Some languages have been invented for special applications, and others for general use. LISP, for example, is widely used by researchers in artificial intelligence — in a sense, for trying to teach a computer how to "think." LOGO was specially designed to bring computer programming to children. Pascal has become the standard language for many university researchers, while Ada is a complex, committee-built language for Department of Defense use.

"Ada is the most complicated language we've got now," Backus says. "It will do a lot of things for you, if you are diligent enough to learn all thousand pages of the manual and can find the features that you want to use."

Backus isn't happy with the restrictions that most of these languages impose. "Conventional languages create unnecessary confusion in the way we think about programs," he says. In attempting to correct the faults of FORTRAN and its descendants, Backus is one of several computer scientists who are determined to add yet another computer language to the list.

Several years ago, Backus outlined the motivation for his present search for a more efficient programming language. "The complacent acceptance most of us give to these enormous, weak languages has puzzled and disturbed me for a long time," he said. "I have tried to analyze some of the basic defects of conventional languages and show that those defects cannot be resolved unless we discover a new kind of language framework."

Backus's approach to creating a new programming language is very mathematical. His "functional" language is almost like algebra — very logical and carefully constructed according to well defined rules. In this language, new procedures can be built from ones already defined. Most of all, this new style of computer language does not require the programmer to spell out in stupefying detail every instruction in a program.

Elliot Soloway of Yale University in New Haven, Conn., is also looking for a better computer language, but he spends his time studying how beginners actually write programs. He and his group in Yale's computer science department are looking particularly at the kinds of mistakes novice programmers make while learning how to program in the language Pascal. From these observations, they hope to come up with a new computer language that better fits the "natural" problem-solving plans that people apparently bring to computer programming.

Soloway says, "Bugs [errors in computer programs] illuminate what a novice is actually thinking — providing us a window on the difficulties as they are experienced by the novice." He continues, "I [would like to] design language constructs that mirror the way they want to think about a problem."

Soloway's scheme for a new programming language will be based on what he learns about the mental models people bring to programming. He disagrees strongly with Backus's approach. "If you make programming like algebra, then people will learn as much about programming as they learn about algebra," Soloway says. "Not a hell of a lot."

In contrast, Backus contends, "Programming languages essentially prescribe a way to think about describing a process. We find that mathematics and orderly thinking of that sort is much more helpful than human factors."

Although Backus and Soloway disagree fundamentally on how to go about designing a new programming language, they are both trying, in their ways, to make programming easier and more accessible. Soloway says, "While we do not believe that everyone need become a professional programmer, it is increasingly important to be able to describe to the computer how it is supposed to realize one's intentions."

Soloway argues, "For such casual programmers, initial difficulties in learning a

programming language may become a permanent barrier to their continuing interaction with computers."

Backus sees his programming language as a way of simplifying and speeding up the process of writing a computer program. "Writing a program is a hideously complex process now," he says, "and we're trying to make it a lot simpler." That means creating a language that is much more orderly and mathematically straightforward, he argues.

"Our hope is that the actual user can write his own programs by taking relatively sophisticated programs that have been written for him and then combining them in such a way that the result does what he wants," Backus says. In most currently used languages, one program usually cannot be arbitrarily tacked onto another program to create a new one. For example, it is generally difficult to stick a computer program that plots graphs into another program, written independently, that does something else.

Backus explains that with a functional language, a user could, for instance, choose three programs that already exist and select two different mathematical operations to combine them. The result would be a new program that in a conventional language would have had to be written as one unit.

"Our goal is to produce a functional language ... so that you can run functional programs on personal computers," says Backus. "We will try to make it widely available so that a lot of people can test the hypothesis that it will really simplify programming." A version of the language optimized for the IBM personal computer may be ready within a few years. Computer languages like LISP already contain some features of a functional programming language.

Soloway's language is at least five to 10 years away. Meanwhile, his research is providing insights into how people program. By studying student programming errors, Soloway has identified about a dozen types of errors that come up over and over again very predictably. Soloway suggests that these errors result because the concepts that novice programmers bring to their task conflict with the way that the computer language requires them to write a program.

For example, something as simple as finding the average of a set of numbers presents difficulties. A program's calculation of an average is usually performed in a loop that reads in a number, adds it to the running total, then reads in another number, and so on, until the process is completed. A second program statement contains a counter to keep track of how many numbers are added.

Soloway says that novices rarely have trouble with the counter because it corresponds to the way people count a sequence of numbers, that is, one by one. However, people normally find a total by writing down all the numbers, then adding them up. "The way the programming language requires you to do a total is different from the model students have," Soloway says. "Thus, they make bugs." His research has identified at least a dozen such trouble spots, he says.

The conclusions that Soloway draws are controversial. Some critics reject the idea that the mind has "natural" or preferred problem-solving strategies. Soloway admits that no one really knows the origin of the "natural" schemes that he hypothesizes people bring to programming. These patterns may be "hard-wired" into the brain, or they may be the result of early learning during childhood.

Mathematician Robert B. Davis of the University of Illinois at Urbana-Champaign has applied research techniques very similar to those used by Soloway. Davis is studying problem-solving by looking at the errors students make while doing arithmetic problems. His results appear to reinforce Soloway's work. Davis says, "Certainly, our evidence seems to show that there are lines of reasoning that people are very comfortable with."

Although in the long term, Soloway's group is working on the design of a new language, in the short term the researchers are developing a computer tutor to help novices overcome some of the obstacles they face when writing programs for the first time.

The latest version of their tutor, named Proust, is designed around the idea of "the remembrance of bugs past." It works by trying to reconstruct how a student created a particular mistake-ridden program from what it "remembers" of its own errors in generating an identical program. Because Proust "knows" the right way to do the program and compares this to what it thinks the student did to write the incorrect program, it can make suggestions. In a sense, Proust behaves like a classroom teacher. By going through this process, Proust theoretically can pick out nonsyntactic errors that occur when computer programs run but still do the wrong thing.

Soloway says, "We assume that the trace Proust generates in recreating this 'buggy' program is the one in the student's head, and then we can tutor with respect to that." He cautions, "That is what Proust is trying to do. Whether we'll be successful in general is another matter."

Soloway accepts the fact that languages like Pascal will be around for a long time to come. Good tutors, whether human or computer, that understand how programming problems arise are needed to help novices get started.

Like Backus, Soloway believes that "we're stuck with lots of conventional languages" because "that's the way it has always been done." Soloway also contends, "We have made a commitment to a certain style of programming for which there is no empirical basis whatsoever."

However, Soloway and Backus disagree on what an alternative language should be like. For beginners, Soloway says, seeing the step-by-step execution of a program as opposed to being able to perform an operation in one go is more important. A functional language would hide the steps making it more difficult for users to understand what is going on.

Backus, on the other hand, says that "word-at-a-time" thinking is merely the result of an "intellectual bottleneck" imposed by the limitations of early computer designs. It prevents us from thinking in terms of larger conceptual units for the task at hand.

Backus says, "While it is perhaps natural and inevitable that languages like FORTRAN and its successors should have developed ... as they did, the fact that such languages have dominated our thinking for 20 years is unfortunate. It is unfortunate because their long-standing familiarity will make it hard for us to understand and adopt new programming styles, which one day will offer far greater intellectual and computational power."

Seymour Papert, creator of LOGO, echoes a similar sentiment in his book *Mindstorms*. He notes as an analogy, that the original typewriter keyboard (the QWERTY arrangement of characters) was the result of an early technical problem. Because adjacent keys often jammed, frequently used letters were deliberately separated. More efficient keyboards with different letter arrangements are available now, but very few people use them.

Papert writes, "There is a tendency for the first usable, but still primitive, product of a new technology to dig itself in.... I think we are well on the road to doing exactly the same thing with the computer." This applies not only to how the computer is used but also to programming languages. Papert goes one step further: "A programming language is like a natural, human language in that it favors certain metaphors, images and ways of thinking. The language used strongly colors the computer culture."

Davis says, "That raises one of my concerns about the current rush to put computers into the classroom and into the home. A lot of habits are being created now that are not necessarily good for the long term."

In the meantime, until something better does come along, novices like this writer have to struggle with languages such as BASIC. My present struggle is practically over, I hope. The program that I started to write hours ago is running, generating one graph after another. But the results don't look right. Is it because of a diabolically hidden error that I haven't found yet or some feature of BASIC that I've unconsciously tapped, or is it because I've stumbled upon some new mathematical result? Help! □