Masterminds of Programming by Chromatic, Federico Biancuzzi

Chapter 4. Forth

Forth is a stack-based, concatenative language designed by Chuck Moore in the 1960s. Its main features are the use of a stack to hold data, and words that operate on the stack, popping arguments and pushing results. The language itself is small enough that it runs on anything from embedded machines to supercomputers, and expressive enough to build useful programs out of a few hundred words. Successors include Chuck Moore's own colorForth, as well as the Factor programming language.

The Forth Language and Language Design

How do you define Forth?

Chuck: Forth is a computer language with minimal syntax. It features an explicit parameter stack that permits efficient subroutine calls. This leads to postfix expressions (operators follow their arguments) and encourages a highly factored style of programming with many short routines sharing parameters on the stack.

I read that the name Forth stands for fourth-generation software. Would you like to tell us more about it?

Chuck: Forth is derived from "fourth," which alludes to "fourth-generation computer language." As I recall, I skipped a generation. FORTRAN/COBOL were first-generation languages; Algol/Lisp, second. These languages all emphasized syntax. The more elaborate the syntax, the more error checking is possible. Yet most errors occur in the syntax. I determined to minimize syntax in favor of semantics. And indeed, Forth words are loaded with meaning.

You consider Forth a language toolkit. I can understand that view, given its relatively simple syntax compared to other languages and the ability to build a vocabulary from smaller words. Am I missing anything else?

Chuck Moore: No, it's basically the fact that it's extremely factored. A Forth program consists of lots of small words, whereas a C program consists of a smaller number of larger words.

By small word, I mean one with a definition typically one line long. The language can be built up by defining a new word in terms of previous words and you just build up that hierarchy until you have maybe a thousand words. The challenge there is 1) deciding which words are useful, and 2) remembering them all. The current application I'm working on has a thousand words in it. And I've got tools for searching for words, but you can only search for a word if you remember that it exists and pretty much how it's spelled.

Now, this leads to a different style of programming, and it takes some time for a programmer to get used to doing it that way. I've seen a lot of Forth programs that look very much like C programs transliterated into Forth, and that isn't the intent. The intent is to have a fresh start. The other interesting thing about this toolkit, words that you define this way are every bit as efficient or significant as words that are predefined in the kernel. There's no penalty for doing this.

Does the externally visible structure consisting of many small words derive from Forth's implementation?

Chuck: It's a result of our very efficient subroutine call sequences. There's no parameter passing because the language is stack-based. It's merely a subroutine call and return. The stack is exposed. The machine language is compiled. A switch to and from a subroutine is literally one call instruction and one return instruction. Plus you can always reach down into the equivalent of an assembly language. You can define a word that will execute actual machine instructions instead of subroutine calls, so you can be as efficient as any other language, maybe more efficient than some.

You don't have the C calling overhead.

Chuck: Right. This gives the programmer a huge amount of flexibility. If you come up with a clever factoring of a problem, you can not only do it efficiently, you can make it extraordinarily readable.

On the other hand, if you do it badly, you can end up with code that no one else can read—code your manager can't understand, if managers can understand anything. And you can create a real mess. So it's a two-edged sword. You can do very well; you can do very badly.

What would you say (or what code would you show) to a developer who uses another programming language to make him interested in Forth?

Chuck: It is very hard to interest an experienced programmer in Forth. That's because he has invested in learning the tools for his language/operating system and has built a library appropriate for his applications. Telling him that Forth would be smaller, faster, and easier is not persuasive compared to having to recode everything. A novice programmer, or an engineer needing to write code, doesn't face that obstacle and is much more receptive—as might be the experienced programmer starting a new project with new constraints, as would be the case with my multicore chips.

You mentioned that a lot of Forth programs you've seen look like C programs. How do you design a better Forth program?

Chuck: Bottom-up.

First, you presumably have some I/O signals that you have to generate, so you generate them. Then you write some code that controls the generation of those signals. Then you work your way up until finally you have the highest-level word, and you call it go and you type `go` and everything happens.

I have very little faith in systems analysts who work top-down. They decide what the problem is and then they factor it in such a way that it can be very difficult to implement.

Domain-driven design suggests describing business logic in terms of the customer's vocabulary. Is there a connection between building up a vocabulary of words and using the terms of art from your problem domain?

Chuck: Hopefully the programmer knows the domain before he starts writing. I would talk to the customer. I would listen to the words he uses and I would try to use those words so that he can understand what the program's doing. Forth lends itself to this kind of readability because it has postfix notation.

If I was doing a financial application, I'd probably have a word called "percent." And you could say something like "2.03 percent". And the argument's percent is 2.03 and everything works and reads very naturally.

How can a project started on punch cards still be useful on modern computers in the Internet era? Forth was designed on/for the IBM 1130 in 1968. That it is the language of choice for parallel processing in 2007 is surely amazing.

Chuck: It has evolved in the meantime. But Forth is the simplest possible computer language. It places no restrictions upon the programmer. He/she can define words that succinctly capture aspects of a problem in a lean, hierarchical manner.

Do you consider English readability as a goal when you design programs?

Chuck: At the very highest level, yes, but English is not a good language for description or functionality. It wasn't designed for that, but English does have the same characteristic as Forth in the sense that you can define new words.

You define new words by explaining what they are in previously defined words mostly. In a natural language, this can be problematic. If you go to a dictionary and check that out, you find that often the definitions are circular and you don't get any content.

Does the ability to focus on words instead of the braces and brackets syntax you might have in C make it easier to apply good taste to a Forth program?

Chuck: I would hope so. It takes a Forth programmer who cares about the appearance of things as opposed merely to the functionality. If you can achieve a sequence of words that flow together, it's a good feeling. That's really why I developed colorForth. I became annoyed at the syntax that was still present in Forth. For instance, you could limit a comment by having a left parenthesis and a right parenthesis.

I looked at all of those punctuation marks and said, "Hey, maybe there's a better way." The better way was fairly expensive in that every word in the source code had to have a tag attached to it, but once I swallowed that overhead, it became very pleasant that all of those funny little symbols went away and were replaced by the color of the word which was, to me, a much gentler way of indicating functionality.

I get interminable criticism from people who are color blind. They were really annoyed that I was trying to rule them out of being programmers, but somebody finally came up with a character set distinction instead of a color distinction, which is a pleasant way of doing it also.

The key is the four-bit tag in each word, which gives you 16 things that we're to do, and the compiler can determine immediately what's intended instead of having to infer it from context.

Second- and third-generation languages embraced minimalism, for example with meta-circular bootstrapping implementations. Forth is a great example of minimalism in terms of language concepts and the amount of hardware support required. Was this a feature of the times, or was it something you developed over time?

Chuck: No, that was a deliberate design goal to have as small a kernel as possible. Predefine as few words as necessary and then let the programmer add words as he sees fit.

The prime reason for that was portability. At the time, there were dozens of minicomputers and then there became dozens of microcomputers. And I personally had to put Forth on lots of them.

I wanted to make it as easy as possible. What happens really is there might be a kernel with 100 words or so that is just enough to generate a—I'll call it an operating system, but it's not quite—that has another couple hundred words. Then you're ready to do an application.

I would provide the first two stages and then let the application programmers do the third, and I was usually the application programmer, too. I defined the words I knew were going to be necessary. The first hundred words would be in machine language probably or assembler or at least be dealing directly with the particular platform. The second two or three hundred words would be high-level words, to minimize machine dependence in the lower, previously defined level. Then the application would be almost completely machine independent, and it was easy to port things from one minicomputer to another.

Were you able to port things easily above that second stage?

Chuck: Absolutely. I would have a text editor, for instance, that I used to edit the source code. It would usually just transfer over without any changes.

Is this the source of the rumor that every time you ran across a new machine, you immediately started to port Forth to it?

Chuck: Yes. In fact, it was the easiest path to understanding how the machine worked, what its special features were based on how easy it was to implement the standard package of Forth words.

How did you invent indirect-threaded code?

Chuck: Indirect-threaded code is a somewhat subtle concept. Each Forth word has an entry in a dictionary. In direct-threaded code, each entry points to code to be executed when that word is encountered. Indirect-threaded code points to a location that contains the address of that code. This allows information besides the address to be accessed—for instance, the value of a variable.

This was perhaps the most compact representation of words. It has been shown to be equivalent to both direct-threaded and subroutine-threaded code. Of course these concepts and terminology were unknown in 1970. But it seemed to me the most natural way to implement a wide variety of kinds of words.

How will Forth influence future computer systems?

Chuck: That has already happened. I've been working on microprocessors optimized for Forth for 25 years, most recently a multicore chip whose cores are Forth computers.

What does Forth provide? As a simple language, it allows a simple computer: 256 words of local memory; 2 push-down stacks; 32 instructions; asynchronous operation; easy communication with neighbors. Small and low-power.

Forth encourages highly factored programs. Such are well-suited to parallel processing, as required by a multicore chip. Many simple programs encourage thoughtful design of each. And requiring perhaps only 1% the code that would otherwise be written.

Whenever I hear people boasting of millions of lines of code, I know they have greviously misunderstood their problem. There are no contemporary problems requiring millions of lines of code. Instead there are careless programmers, bad managers, or impossible requirements for compatibility.

Using Forth to program many small computers is an excellent strategy. Other languages just don't have the modularity or flexibility. And as computers get smaller and networks of them are cooperating (smart dust?), this will be the environment of the future.

This sounds like one major idea of Unix: multiple programs, each doing just one thing, that interact. Is that still the best design today? Instead of multiple programs on one computer, might we have multiple programs across a network?

Chuck: The notion of multithreaded code, as implemented by Unix and other OSes, was a precursor to parallel processing. But there are important differences.

A large computer can afford the considerable overhead ordinarily required for multithreading. After all, a huge operating system already exists. But for parallel processing, almost always the more computers, the better.

With fixed resources, more computers mean smaller computers. And small computers cannot afford the overhead common to large ones.

Small computers will be networked, on chip, between chips and across RF links. A small computer has small memory. Nowhere is there room for an operating system. The computers must be autonomous, with a self-contained ability to communicate. So communication must be simple—no elaborate protocol. Software must be compact and efficient. An ideal application for Forth.

Those systems requiring millions of lines of code will become irrelevant. They are a consequence of large, central computers. Distributed computation needs a different approach.

A language designed to support bulky, syntactical code encourages programmers to write big programs. They tend to take satisfaction, and be rewarded, for such. There is no pressure to seek compactness.

Although the code generated by a syntactic language might be small, it usually isn't. To implement the generalities implied by the syntax leads to awkward, inefficient object code. This is unsuitable for a small computer. A well-designed language has a one-one correlation between source code and object code. It's obvious to the programmer what code will be generated from his source. This provides its own satisfaction, is efficient, and reduces the need for documentation.

Forth was designed partly to be compact in both source and binary output, and is popular among embedded developers for that reason, but programmers in many other domains have reasons to choose other languages. Are there aspects of the language design that add only overhead to the source or the output?

Chuck: Forth is indeed compact. One reason is that it has little syntax.

Other languages seem to have deliberately added syntax, which provides redundancy and offers opportunity for syntax checking and thus error detection.

Forth provides little opportunity for error detection due to its lack of redundancy. This contributes to more compact source code.

My experience with other languages has been that most errors are in the syntax. Designers seem to create opportunity for programmer error that can be detected by the compiler. This does not seem productive. It just adds to the hassle of writing correct code.

An example of this is type checking. Assigning types to various numbers allows errors to be detected. An unintended consequence is that programmers must work to convert types, and sometimes work to evade type checking in order to do what they want.

Another consequence of syntax is that it must accommodate all intended applications. This makes it more elaborate. Forth is an extensible language. The programmer can create structures that are just as efficient as those provided by the compiler. So all capabilities do not have to be anticipated and provided for.

A characteristic of Forth is its use of postfix operators. This simplifies the compiler and offers a one-one translation of source code to object code. The programmer's understanding of his code is enhanced and the resulting compiled code is more compact.

Proponents of many recent programming languages (notably Python and Ruby) cite readability as a key benefit. Is Forth easy to study and maintain in relation to those? What can Forth teach other programming languages in terms of readability?

Chuck: Computer languages all claim to be readable. They aren't. Perhaps it seems so to one who knows the language, but a novice is always bewildered.

The problem is the arcane, arbitrary, and cryptic syntax. All the parentheses, ampersands, etc. You try to learn why it's there and eventually conclude there's no good reason. But you still have to follow the rules.

And you can't speak the language. You'd have to pronounce the punctuation like Victor Borgia.

Forth alleviates this problem by minimizing the syntax. Its cryptic symbols `@` and `!` are pronounced "fetch" and "store." They are symbols because they occur so frequently.

The programmer is encouraged to use natural-language words. These are strung together without punctuation. With good choice of words, you can construct reasonable sentences. In fact, poems have been written in Forth.

Another advantage is postfix notation. A phrase like "6 inches" can apply the operator "inches" to the parameter 6, in a very natural manner. Quite readable.

On the other hand, the programmer's job is to develop a vocabulary that describes the problem. This vocabulary can get to be quite large. A reader has to know it to find the program readable. And the programmer must work to define helpful words.

All in all, it takes effort to read a program. In any language.

How do you define success in terms of your work?

Chuck: An elegant solution.

One doesn't write programs in Forth. Forth is the program. One adds words to construct a vocabulary that addresses the problem. It is obvious when the right words have been defined, for then you can interactively solve whatever aspect of the problem is relevant.

For example, I might define words that describe a circuit. I'll want to add that circuit to a chip, display the layout, verify the design rules, run a simulation. The words that do these things form the application. If they are well chosen and provide a compact, efficient toolset, then I've been successful.

Where did you learn to write compilers? Was this something everybody at the time had to do?

Chuck: Well, I went to Stanford around '60, and there was a group of grad students writing an ALGOL compiler—a version for the Burroughs 5500. It was only three or four of them, I think, but I was impressed out of my mind that three or four guys could sit down and write a compiler.

I sort of said, "Well, if they can do it, I can do it," and I just did. It isn't that hard. There was a mystique about compilers at the time.

There still is.

Chuck: Yeah, but less so. You get these new languages that pop up from time to time, and I don't know if they're interpreted or compiled, but well, hacker-type people are willing to do it anyway.

The operating system is another concept that is curious. Operating systems are dauntingly complex and totally unnecessary. It's a brilliant thing that Bill Gates has done in selling the world on the notion of operating systems. It's probably the greatest con game the world has ever seen.

An operating system does absolutely nothing for you. As long as you had something—a subroutine called disk driver, a subroutine called some kind of communication support, in the modern world, it doesn't do anything else. In fact, Windows spends a lot of time with overlays and disk management all stuff like that which are irrelevant. You've got gigabyte disks; you've got megabyte RAMs. The world has changed in a way that renders the operating system unnecessary.

What about device support?

Chuck: You have a subroutine for each device. That's a library, not an operating system. Call the ones you need or load the ones you need.

How do you resume programming after a short hiatus?

Chuck: I don't find a short coding hiatus at all troublesome. I'm intensely focused on the problem and dream about it all night. I think that's a characteristic of Forth: full effort over a short period of time (days) to solve a problem. It helps that Forth applications are naturally factored into subprojects. Most Forth code is simple and easy to reread. When I do really tricky things, I comment them well. Good comments help re-enter a problem, but it's always necessary to read and understand the code.

What's the biggest mistake you've made with regard to design or programming? What did you learn from it?

Chuck: Some 20 years ago I wanted to develop a tool to design VLSI chips. I didn't have a Forth for my new PC, so I thought I'd try a different approach: machine language. Not assembler language, but actually typing the hex instructions.

I built up the code as I would in Forth, with many simple words that interacted hierarchically. It worked. I used it for 10 years. But it was difficult to maintain and document. Eventually I recoded it in Forth and it became smaller and simpler.

My conclusion was that Forth is more efficient than machine language. Partly because of its interactivity and partly because of its syntax. One nice aspect of Forth code is that numbers can be documented by the expression used to calculate them.

With Safari, you learn the way you learn best. Get unlimited access to videos, live online training, learning paths, books, interactive tutorials, and more.

Start Free Trial