(Approx. 986 words)

## Forth Programs

By Dick Maybach, Brookdale Computer User Group

www.bcug.com

n2nd (at) att.net

As computers have evolved, they have become remarkably cheap, which has affected how we use them and how their software is developed. In the late 60s, minicomputers were beginning to appear in smaller companies, often as controllers, and these had, by today's standards, tiny amounts of RAM, often around eight kilobytes. Then, one byte of RAM costs about a dollar, which means that a GigaByte of it would have cost about a billion dollars. As a result, minimizing memory use was often a key objective for programs and the tools to create them. The Forth language, <https://en.wikipedia.org/wiki/Forth_(programming_language)>, was developed in this environment. Compared to modern languages, writing programs in it was tedious and error-prone. Nevertheless, let's briefly look at it, if for nothing else than to appreciate how far we've come. If you would like to experiment, development software for Gforth, a modern variant, is available for most platforms at <https://gforth.org/>, as is an online manual.

Forth uses reverse-Polish arithmetic, which is best explained by example. Operators separate operands in far more common arithmetic notation, and an equals sign causes the operation to occur.

2 + 4 =

In reverse-Polish notation, the operands appear first, followed by the operator. There is no equal sign as the operation occurs immediately.

2 4 +

A key concept here is the stack, which acts as temporary storage. In the above example, the 2 is pushed onto the stack; then the 4 is pushed. Finally, the + operator pops the two operands off the stack, sums them, and pushes the result back onto the stack. Figure 1 shows an example. The ".s" command displays the number of items on the stack (inside < >) and its contents. An "ok" indicates the operation was successful. Items in Forth are separated by whitespace (one or more spaces or tabs). A carriage return causes the entire line to be executed.

|  |  |  |
| --- | --- | --- |
| Command | Output | Comment |
| .s | <0> ok | The stack is empty. |
| 2 4 6 .s | <3> 2 4 6 ok | 2, 4, 6 are pushed onto the stack. |
| + .s | <2> 2 10 ok | The two most recent stack items are popped, and their sum pushed. |
| \* .s | <1> 20 ok | The two most recent stack items are popped and their product pushed. |

Figure 1. Forth Code Example.

In Figure 1, a carriage return was entered after every ".s," which causes the result and "ok" to be added to the end of that same line. Figure 2 shows a screenshot of the session running this example.

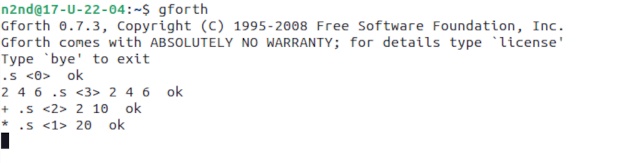


Figure 2. Screenshot of a Forth Session.

Like most languages, Forth supports functions. For example, figure 3 shows a session that defines three functions (called "words" in Forth-speak) square, cube, and fourth.

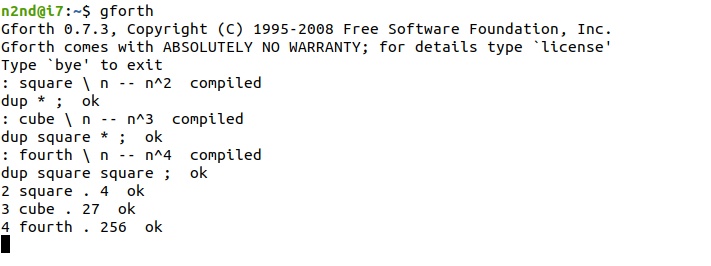


Figure 3. Forth Word (Function) Definition and Use.

Each definition begins with a colon and ends with a semicolon. A backslash starts a comment, ending at the line's end. Finally, the program adds "compiled" to show that each line is valid and "ok" at the end of the definition. The forms of the comments are standard Forth usage. For example, "n – n^2" signifies that a number on the stack is replaced by its square. Note the "." command in the lines that invoke the functions, which pops the last number off the stack and prints it.

Unlike most modern languages, Forth doesn't access variables by name. Instead, it pulls them off the stack, which is constantly changing. So, for example, "dup" duplicates the last variable, and we calculate the square by duplicating and multiplying. Of course, for all but trivial functions, it's more efficient to use local variables, but I'll omit the details in this introduction to the language.

In the above examples, we're using Forth as a calculator; it discards everything when we close the program. You can store function definitions in text files for later use, as shown in Figure 4.

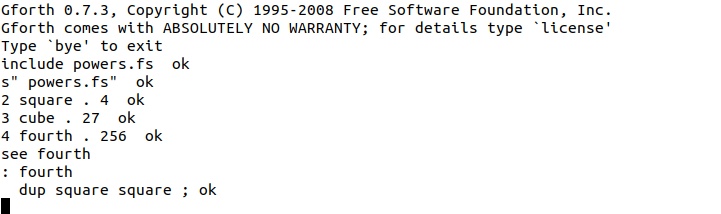


Figure 4. Reading Functions from a File.

The definitions of square, cube, and fourth had been stored in the text file "powers.fs." In this session, the first two lines read and execute that file. The last three lines perform as in Figure 3, where the same functions were defined during that session. Note the line "see fourth," which displays the definition of that word. Programmers often use the see command to check the functions they've defined.

So far, all the examples have used integer arithmetic. Although it's seldom used today, years ago, many small computers could not process in floating point, and if they did, they stored floating-point numbers separately from integers and used different functions on them. Forth preserves this distinction with a separate stack for floating point quantities. Figure 5 compares the two types.

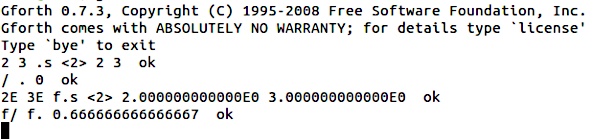


Figure 5. Integer and Floating-point Operations.

The first command line pushed the integers 2 and 3 onto the stack, and the second divides them and pops the result, which is zero. The third pushes the numbers 2E and 3E onto the stack. (The Es force these to be floating-point numbers.) The operation "f.s" displays the floating-point stack. The fourth command line performs a floating-point division and pops the result of the floating-point stack. In Forth, operation names beginning with "f" are floating-point ones.

Forth is a difficult language to use. It's easy to lose track of the contents of the stacks. You can use the see command to show the code associated with a word, but the comments are stripped, which means you must make careful records. As a result of confusing syntax, most words are just a few lines, but this increases the number of functions, making it difficult to track what each does. As computers became more powerful, program languages were developed that were far easier to use. We now see Forth as a historical artifact with little purpose other than to show the environment with which early developers had to contend.