My Forth Tech Talk

Simon Parent

Thursday April 19, 2012

Forth

Forth is a programming language which was created by Chuck Moore in 1970.

Well, at least that's when its defining features had mostly settled into place. Forth itself had simply evolved from Chuck Moore's personal programming environment (which he brought with him from job to job) which dates back to 1958.

I will begin by giving an overview of Forth's superficial features.

Essentially No Syntax

The Forth interpreter is the following very simple loop:

- Scan the input until the next space character is found.
- The characters we read up to the space are a **word**.
- Execute the command corresponding to that word.

Thus, Forth code is simply a sequence of commands (words).

Essentially No Syntax

The Forth interpreter is the following very simple loop:

- Scan the input until the next space character is found.
- The characters we read up to the space are a **word**.
- Execute the command corresponding to that word.

Thus, Forth code is simply a sequence of commands (words).

What about string literals? The standard Forth solution is just to have a word named ", which reads from the input until the next ". Thus, when it returns to the interpreter, the input pointer will have been advanced just past the end of the string literal.

```
" HELLO WORLD" DISPLAY
```

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2 2 3 + + 6 * DISPLAY
```

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
|2 2 3 + + 6 * DISPLAY
```

Stack: (empty)

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

2 | `2 3 + + 6 * DISPLAY`

Stack: 2 END

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2  2  3 + + 6 * DISPLAY
```

Stack: 2 2 END

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2 2 3 + + 6 * DISPLAY
```

Stack: 2 2 3 END

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2 2 3 + + 6 * DISPLAY
```

Stack: 4 3 END

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2 2 3 + + |6 * DISPLAY
```

Stack: 7 END

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2 2 3 + + 6 * DISPLAY
```

Stack: 6 7 END

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2 2 3 + + 6 * DISPLAY
```

Stack: 42 END

The Parameter Stack (Reverse Polish Notation)

Let's calculate (and display)

$$(2 + 2 + 3) \times 6$$

The Forth code would be:

```
2 2 3 + + 6 * DISPLAY
```

Stack: END
Display: 42

A Typical Implementation

```
while(true) {
  scanf("%s", word);
  if(strcmp(word, "+") == 0) {
    int a = pop(); int b = pop();
    push(a + b);
  } else if(strcmp(word, "DISPLAY") == 0) {
    int a = pop();
    printf("%d ", a);
  } else if( ···
    ··· other commands ···
  } else {
    if(!is_number(word))
      (ERROR)
    push(atoi(word));
  }
}
```

A Better Implementation

```
while(true) {
  scanf("%s", word);
  if(entry = lookup(word)) {
    execute(entry.code);
  } else {
    if(!is_number(word))
      (ERROR)
    push(atoi(word));
  }
}
```

Defining New Words, and the Return Stack

Now that the **dictionary** of words is dynamic, we may get the idea of allowing new words to be added at run-time.

```
: NORM-SQUARED DUP * SWAP DUP * + ;
```

e.g. `3 4 NORM-SQUARED` produces 25 on the stack.

Now our interpreter needs to keep track of return addresses. Forth uses a separate return stack for this. This is a very unique feature; almost every other programming environment has a single stack in which data and return addresses are interleaved.

Forth is also a Compiler

`:` does not just define a new word, it compiles the new word.

Our memory model is as follows. There is a (data) stack, and a return stack. There is a large area of memory (the data space) where we store the dictionary, as a linked list of word entries. There is always a pointer to the end of this area of memory, and we can append the data for new entries to it.

Like `"`, `:` takes over from the interpreter in order to do its job. It reads the next word from input, and uses this as the name for the definition, writing out the header information to the data space. It then reads words until it encounters `;`. Whereas the interpreter would execute the word, it finds the word in the dictionary, and then appends the address of its dictionary entry to the data space.

Forth is also a Compiler

Actually, there are a few subtle issues to deal with, but this is the basic idea.

Note that we only perform name lookup for a word when we are compiling it for the first time. So, the choice of a data structure that requires linear lookup is actually not so bad.

Forth is also a Compiler

Actually, there are a few subtle issues to deal with, but this is the basic idea.

Note that we only perform name lookup for a word when we are compiling it for the first time. So, the choice of a data structure that requires linear lookup is actually not so bad.

Also note that we can define new defining words. For example, it would be nice to have a word that lets us define a global variable. One way this could work is that `VARIABLE FOO` allocates enough room in the data space (by appending zeroes to it, say), and then also appends the definition of a word `FOO` which returns the address of that allocated space.

Philosophy of Forth

The normal approach to software development involves first creating many of these (and possibly others) as separate programs. Then, we can use this stack of software to create a program which solves our actual problem.

- assembler
- compiler
- file system
- operating system
- interactive shell
- text editor

Philosophy of Forth

Forth is a much better "high-level assembly language" than C.

```
355 113 */
```

Philosophy of Forth

The Basic Principle, according to Chuck Moore:

Keep it Simple

Philosophy of Forth

The Basic Principle, according to Chuck Moore:

Keep it Simple

The Basic Principle has a corollary:

Do Not Speculate!

Philosophy of Forth

The Basic Principle, according to Chuck Moore:

Keep it Simple

The Basic Principle has a corollary:

Do Not Speculate!

Another, extremely controversial corollary:

Do It Yourself!

Philosophy of Forth

"But suppose everyone wrote their own subroutines? Isn't that a step backward; away from the millennium when our programs are machine independent, when we all write in the same language, maybe even on the same computer? Let me take a stand: I can't solve the problems of the world. With luck, I can write a good program." – Chuck Moore

Forth as your Assembler

To aid in the bootstrapping process, once the initial kernel has been written for the target machine, the next thing you want is some words which can help you emit machine code.

Implementing this in a natural way will cause the assembler syntax itself to become more Forthy. For example, code to emit `xor eax, eax` might be something like:

```
EAX EAX XOR ,
```

Since all Forth words receive arguments and return values using the data stack, even Forth words implemented in machine code have a machine-independent interface.

Forth as your File System

Forth divides the available disk space into **blocks**, which are traditionally 1K (1024 bytes) each. They are simply numbered starting from 0.

There are no files, no directories, or even file names. Although there is nothing stopping you from using constants to replace the block number references in your code.

Also, it is probably worth writing at each some functions to manage a list of the unused blocks.

Forth as your Text Editor

```
0 C1 42 # :R RECORD
```

Forth as your Text Editor

```
0 C1 42 # :R RECORD
```

(*n* `LINE` creates a "pointer" to the *n*th line in the current block.)

```
: LINE
  1 - 7 * RECORD + ;
```

Forth as your Text Editor

```
0 C1 42 # :R RECORD
```

(*n* `LINE` creates a "pointer" to the *n*th line in the current block.)

```
: LINE
  1 - 7 * RECORD + ;
```

(Command to "type" (print) a line to the screen. For example, `4 R` will display the fourth line.)

```
: T
  CR LINE ,C ;
```

Forth as your Text Editor

```
0 C1 42 # :R RECORD
```

(*n* `LINE` creates a "pointer" to the *n*th line in the current block.)

```
: LINE
  1 - 7 * RECORD + ;
```

(Command to "type" (print) a line to the screen. For example, `4 R` will display the fourth line.)

```
: T
  CR LINE ,C ;
```

(Command to replace a line. For example, `" HELLO" 6 R` will replace the sixth line with `HELLO`.)

```
: R
  LINE =C ;
```

Forth as your Text Editor

```
( Display all lines in the current block. )
: LIST
  15 0 DO 1 +
    CR DUP LINE ,C DUP ,I
  CONTINUE ;
```

Forth as your Text Editor

```
( Display all lines in the current block. )
: LIST
  15 0 DO 1 +
    CR DUP LINE ,C DUP ,I
  CONTINUE ;

( Insert a line of text. )
: I
  1 + DUP 15 DO 1 -
    DUP LINE DUP 7 + =C
  CONTINUE R ;

( Delete a line of text. )
: D
  15 SWAP DO 1 +
    DUP LINE DUP 7 - =C
  CONTINUE " " 15 R ;
```

Forth as your Operating System

It's not so hard to add co-operative multitasking.

Just modify the initialization code to produce many sets of stacks, and give one out to each task.

Then, create a word `PAUSE`, which saves the current context, and finds the next task which is able to run.

Documentation

Chuck Moore did not write much about documentation when describing his philosophy, except to say

> *"Use comments sparingly! . . . Programs are self-documenting, even assembler programs, with a modicum of help from mnemonics."*

Documentation

Chuck Moore did not write much about documentation when describing his philosophy, except to say

> *"Use comments sparingly! . . . Programs are self-documenting, even assembler programs, with a modicum of help from mnemonics."*

However, in his current Forth system, colorForth, each block of source code has an associated "shadow block", which is expected to contain the documentation for that block of source code.

Relevance Today

For Forth's philosophy to really shine, it has to have complete control of the machine, which is not really practical for the way we use computers today.

Relevance Today

For Forth's philosophy to really shine, it has to have complete control of the machine, which is not really practical for the way we use computers today.

Bootstrapping. You need to "do it yourself" anyway, and it doesn't really matter how, since the stage loaded after you is going to throw it away anyway.

Relevance Today

For Forth's philosophy to really shine, it has to have complete control of the machine, which is not really practical for the way we use computers today.

Bootstrapping. You need to "do it yourself" anyway, and it doesn't really matter how, since the stage loaded after you is going to throw it away anyway.

May be viable for many-CPU computing, as long as the CPUs are independent enough, Especially if we make so many of them that they are very resource constrained individually. Interestingly enough, Chuck Moore is currently the CTO of GreenArrays, Inc., which is pursuing this possibility.

Relevance Today

For Forth's philosophy to really shine, it has to have complete control of the machine, which is not really practical for the way we use computers today.

Bootstrapping. You need to "do it yourself" anyway, and it doesn't really matter how, since the stage loaded after you is going to throw it away anyway.

May be viable for many-CPU computing, as long as the CPUs are independent enough, Especially if we make so many of them that they are very resource constrained individually. Interestingly enough, Chuck Moore is currently the CTO of GreenArrays, Inc., which is pursuing this possibility.

Education? Fun? Historical curiosity?

Questions?

References

- History of Programming Languages, Volume II (1996)
- `http://colorforth.com` (Chuck Moore's personal web site)
  - Programming a Problem-Oriented-Language (1970) (`http://www.colorforth.com/POL.htm`)
- `jonesforth.s` – if you want to read more about implementation details
- `http://www.greenarraychips.com`