FORTH: A Stack Programming Language

Harley Witt

Russell Lewis

May 14th, 2008

Introduction:

Forth is stack based programming language and therefore uses postfix notation. It is composed of numbers and words. In the following statement which adds 1 and 2, “1” and “2” are numbers and “+” is a word.

```
1 2 +
```

The language is interactive and therefore as the user enters “1 2 +” and then hits the enter key, the program begins to interpret left to right, top to bottom. “1” is pushed on the the stack, followed by “2”. Then the “+” is looked up in the dictionary and this function is carried out. The “+” word will take the top two items in the stack, add them and then push the result onto the stack. At this point since “1” and “2” were removed off the stack, the result “3” is at the top of the stack.

Let's examine another simple example to multiply the number 5 by itself (5 * 5) as follows:

```
5 DUP *
```

“5” is pushed onto the stack, then DUP is looked up and executes its function which copies the topmost element on the stack and pushes the result onto the stack. “*” then pops the top 2 elements, multiplies them and pushes the result onto the stack which would be “25”.

The final simple example is a function to determine the absolute value of an item on the stack. This could otherwise be expressed as (i<0) ? -i:i

```
DUP 0 < IF -1 * THEN
```

Let's assume -3 is the top item on the stack. “DUP” will push another -3 onto the stack. “0” gets pushed onto the stack. “<” is looked up and pops 2 elements from the stack (0 and -3), compares them and pushes that resulting boolean value (“T”) onto the stack. “IF” is looked up and pops the “T” from the stack, since the value is “T” the interpreter continues. “-1” is pushed onto the stack. “*” is looked up and pops two items off the stack (-3 and -1), multiplies them and pops the result “3” onto the stack. “THEN” is looked up and no operation is performed. We therefore see the result of and absolute value function that turned the element “-3” into a “3” at the top of the stack.

History:

In the 1960s Charles (Chuck) H. Moore[1] was a scientist working on programming languages such as ALGOL and FORTRAN at MIT and the Stanford Linear Accelerator Center. He felt too much time was spent writing, compiling and running code. He focused his efforts on coming up with a solution to create a text interpreter (written in ALGOL) which was run in an interactive mode and acted upon items that were either numbers or words. This combined the steps of edit, compile and run into command line level interpretation and compilation. Special words were created such as the ":" to mark the beginning of code that should switch the interpreter into compile mode and ";" to mark the switch back to interpreter mode. When such a ":" <name> ... ";" block was encountered the compiler would store the "word" (we would call it a function) into the dictionary for later use during interpreter mode. Writing code utilizing two stacks (one for temporary values and parameters, one for stack frames) and simple dictionary allowed the code size to be compact, efficient and extensible.

In 1970 while working on an IBM 3rd generation computer, he felt as if the language he was writing was a fourth generation language. As the computer only allowed for 5 character names, the result was to name the language "FORTH." The first public version of FORTH was used at the National Radio Astronomy Observatory in Arizona in 1971. The program was used for data acquisition and analysis as well as concurrently controlling the radio telescope. The popularity of the language quickly spread within the Astronomy community. He later spawned initiatives to further promote the FORTH language and many different versions of FORTH were created. MiniFORTH and MicroFORTH were used in minicomputers and micro controllers. The compact size of the programming language images continues to provide value in boot loaders today.

Applications:

FORTH is a strange beast. An imperative, stack-based language, FORTH combines low-level computational capabilities with a remarkably powerful function declaration mechanism. The former allows FORTH programs, when compiled by an optimizing compiler, to run nearly as fast as hand-optimized assembly; the latter allows a FORTH program to literally rewrite the compiler as the program is being compiled and executed.

Because of these two very different strengths, FORTH is used in two very different classes of applications. When an aggressive optimizing compiler is available, FORTH can be useful as a powerful numerical processing language. On the other end of the spectrum, FORTH is useful in embedded systems, on prototype hardware, or as a boot loader, because only the most minimal compiler kernel is required; once a FORTH compiler kernel is available, a fully-functional FORTH compiler can literally be built on the fly using only the FORTH language itself.

Analysis:

The basic FORTH language is essentially a portable wrapper around assembly language, plus the basic mechanisms needed to allow compiler self-modification. FORTH provides just enough abstraction to allow for a type of "structured assembly language;" or, with effort, the function-declaration mechanism can be used to build a complete suite of higher-level language constructs, including a variety of loop types, if/then/else statements, and even function pointers.

However, while FORTH allows for the construction of extremely complex **control** structures, it does not appear that FORTH has any significant capability to implement complex **memory** structures. (If it is possible, the "authoritative book" on FORTH makes no mention of it.) FORTH's memory management is painful in the extreme and astonishingly primitive. The only native storage mechanisms are for single- or double-word values (usually integers); to construct anything as simple as an array, one must allocate a variable and then increment an internal pointer in the dictionary to allocate more space. (Thus, an array in FORTH is simply a region of undocumented memory.) Presumably, records could be created in a similar fashion, but FORTH provides no standard, reliable mechanisms for indexing arrays or accessing member fields. FORTH expects you to simply calculate the indices yourself and increment pointers, just as in typical assembly language programs.

Moreover, FORTH does not provide any native heap. In order to have a heap in FORTH, a programmer must write his own memory management routines, and the memory behind a heap is simply the constituent bytes of an array. Since arrays in FORTH can never be resized, this means that the heap can never expand, severely limiting its utility.

Finally, FORTH is hampered by its own syntax. Remarkably, FORTH uses parentheses for comments, meaning that it is difficult to organize expressions into logical blocks. Certainly, the (postfix-operator) expression

```
2 3 4 / 5 6 7 * + * +
```

is technically unambiguous and "elegantly" stack-oriented, but it is very difficult to read. FORTH would be greatly improved simply by the ability to group expressions in a more conventional fashion. This problem, of course, goes beyond simple arithmetic expressions; the following is valid FORTH code:

```
1 2 3 4 5 6 FOO BAR BAZ
```

(assuming that the "words" FOO BAR BAZ have been defined), but the syntax does not give any way to hint which of the numbers (or words!) could be arguments to which other elements.

In summary, it appears that FORTH may be a very exciting way to write portable assembly language, or might be useful for more complex programs if the programmers were very disciplined to carefully organize and document their code. However, it lacks key features necessary to make it a successful general-purpose language.

A Luca VM Interpreter:

A fun exercise is to blend our LUCA experience together with our newly acquired FORTH knowledge led us to experiment with creating a LUCA VM interpreter in FORTH. The following is an example word to compile into the FORTH dictionary to support interpreting addi. Next we examine required FORTH code to support the following from LUCA:

```
pusha 0 15 0

pushi 0 37

storei 0
```

In FORTH, to support these a word for each will be compiled into FORTH as follows:

```
: pusha
        1 IGNORE
        NUMBER
        GETVARADDR
        1 IGNORE        ;
```

The stack after the “pusha 0 15” now contains (0)

```
:pushi
        1 IGNORE
        NUMBER          ;
```

The stack after the “pushi 0 37” now contains (0 37)

```
:storei
        1 IGNORE
        SWAP
        LUCATOFORTH
        !           ;
```

The stack is now empty.

Concepts:

Is the FORTH language pass by value or pass by reference? Yes it could be. It can be considered either or and even perhaps neither of them. There is only one common parameter stack. The programmer has the option when compiling words into the dictionary to act upon the stack in any way they wish. One method may be to copy the relevant parameters onto the top of the stack before the word that operates on them is interpreted. The word would expect that the parameters will be removed and replaced with resulting values placed on the top of the stack. However, the word (or function) can be written such that it copies the values within the function. These two methods either access the direct parameters (reference) or utilize a copy (value). Perhaps it is more appropriate to state that parameters are not really being passed at all since the parameter stack is one single stack so no passing needs to take place.

FORTH has no built in garbage collection. The Dictionary is one large contiguous area in memory but can even be thought of as a stack itself. As words are compiled into the dictionary they act as if they are put onto the top of the stack. When a word is interpreted, a lookup in the dictionary starts from the top of the stack and searches downward. This means that words can be redefined and the most recent one will be used. There is a very interesting behavior to point out. Consider the case in which a word has been looked up in the dictionary. This word can be composed of other words. However, the words will be looked up only farther down (below) the current word entry. Let's examine the following:

```
4 4 +                \ The result will put 8 on the stack

: add + ;\ This compiles the word “add” into the dictionary which
\ will behave like “+”

: + 5 + ; \ This redefines the word “+” to add 5

4 4 +                \ The latest “+” is used and therefore only one 4 is popped
          \and 5 is added to it.  The resulting stack contains (4 9)

4 4 add \ The “add” is looked up which is composed of “+” but since                    \ the
language only scans down from “add” the previous “+”                    \ is used and 8 will be put on
the stack
```

This example of the dictionary behavior was used to provide on interesting point about the closest ability to not exactly “free” memory but to make already used memory available again. The “FORGET” word when given an argument of a word will essentially reset the dictionary back to the state prior to when that word was compiled. Therefore, a “FORGET +” will result in “+” behaving as it did prior. However, any other compiled words that happened after the last “+” will also be lost.

Strengths:

Bootloaders & Prototype Hardware

The base dictionary of supported words is very small and can be easily extended to support specific functions. This is quite useful for bootloaders and to get prototype hardware running quickly.

"Portable Assembly Language"

Assuming the programmer uses the dictionary wisely and the reuse of base words maintains tight and compact code, the applications can be developed more quickly and with similar speed to that of assembly code. However, with base support across different devices, FORTH can be implemented on many different machines as a type of portable assembly language.

Extreme Self-Reflection And Modification

The structure of FORTH requires small pieces or functions to be compiled into the dictionary. This promotes an efficient methodology for creating code. The smaller functions can be reused frequently and extension to base functions is simple. Note that careful usage and manipulation of the stack is required to avoid corruption of values on the stack. However, the testing of each small function's input and output makes it easy to debug.

Integrated Interpreter/Compiler

Reviewing the history of FORTH, we mentioned that it was created to accelerate and simplify programming. One may begin typing and have each line of code instantly interpreted or compiled. This is a very quick way to write and test applications.

Variables:

Creation

Variables are supported in FORTH and are created by preceding them with the WORD “VARIABLE”.

```
VARIABLE foo                          \ int foo;
```

To create an array of variables one must follow the variable with the WORDS “CELLS ALLOT” as follows:

```
VARIABLE bar 10 CELLS ALLOT           \ int[10] bar;
```

Accessing

To reference the variable, address just the variable name is used:

```
foo                                   \ & foo
```

To read the value of the variable, one must follow it with the “@” symbol:

```
foo @                                 \ “read” foo
```

Assignment

To assign a value to the variable, following the variable by “!”. The value on the stack is used:

```
1 foo !                               \ foo = 1;
```

To assign a value to a location in the array:

```
foo @ bar 3 CELLS + !                 \ bar[3] = foo;
```

Note how complex the code becomes when performing array operations:

```
bar foo @ CELLS + @ bar foo @ 1 – CELLS + !     \ bar[foo-1] = bar[foo];
```

Weaknesses:

The reason variables were chosen to be explained between strengths and weaknesses is to help show a few of the weaknesses of FORTH.

Difficult Syntax

The postfix style is quite different than many languages and is therefore difficult to understand at first. The examples for variables show how difficult and cumbersome working with them can be. The readability of variable manipulation also takes considerable time and extra code to view in order to understand it fully.

Inconsistent syntax

Most of the syntax operates left to right, top down. Words are interpreted and are expected to act upon the stack. However, this is not consistent.

Terrible Memory Management

Perhaps It would be more appropriate to say that there is NO memory management. The dictionary will continue to grow in size and a special word “FORGET” exists to delete all of the items in the dictionary back to the word which is supplied after the “FORGET” word. This could hardly be considered memory management. There is no built in garbage collection. Very complicated routines could be created but the inherent stack based design makes it a complex and time consuming possibility.

No Type Checking

The language is simple and has no type checking. It is the responsibility of the programmer to check the values or assume they are of the correct type. Initially when learning the language the only type explained is the basic integer number. All the rest are covered as variables and ways in which one interacts with non numbers is more complicated.

Conclusion:

FORTH is simple, modular, extensible and interactive. The stack based postfix notation is different than most languages but the experience is similar to learning how to use postfix on a calculator. It is painful at first. However, once learned, the power of the stack is unleashed upon the programmer.

Learning the language for basic programming is initially quite enjoyable. Starting with only Words and Numbers with which the Words are either interpreted or compiled into the dictionary in an interactive mode gives instant feedback to the user. However, when more complicated constructs such as variables need to be used, than the initial euphoria of playing with the new language comes to an abrupt end. Just trying to figure out how to use and manipulate floating point numbers has a steep learning curve.

Initially, the best qualities of FORTH made it a prime candidate for systems with limited memory that needed simple and fast functionality. Many versions of FORTH have been developed and can be run on various hardware. This allows many choices for programmers. However, as the advanced functionality of complex FORTH implementations increases, the reason for initially choosing FORTH begins to diminish. The dictionary becomes larger and the code less efficient and therefore it is hard to say how much like FORTH the forth derived languages truly are.

References

[1] Brodie, Leo (2007). in Marcel Hendrix: Starting Forth (Online book), Marlin Ouverson, Web edition (in English), FORTH, Inc.. Retrieved on 2007-09-29.

[2] M. Anton Ertl. Is Forth Code Compact? A Case Study. In EuroForth'99 Conference Proceedings, St. Petersburg, Russia, 1999.

[3] Brodie, Leo (2004). in Bernd Paysan: Thinking Forth (PDF Online book) (in English). ISBN 0-9764587-0-5. Retrieved on 2006-06-19.

[4] Conklin, Edward K.; Elizabeth D. Rather et al (08). Forth Programmer's Handbook (paperback), 3rd (in English), BookSurge Publishing, 274. ISBN 1-4196-7549-4.

[5] Forth (programming language). (2008, May 14). In Wikipedia, The Free Encyclopedia. Retrieved 07:34, May 14, 2008, from http://en.wikipedia.org/wiki/Forth_(programming_language)