FORTH - A simple stack oriented language

What is Forth?

In a nutshell, Forth is a combination compiler and interpreter. The compiler translate source code not to machine code like we saw in the previous chapter, but into instructions for a "virtual" machine, which we'll refer to as "pcode". The same idea is used in Java and Python and is very in modern dynamic languages.

However, in Forth, we program at this pcode level, at what we might almost call an assembly language for the virtual machine. But it is language that can extended in interesting ways that give it quite a dynamic character.

With Forth you play what I would like to call "snippets of computation", referred to as "words" that manipulate a push down data stack directly or get strung together to build (compile) new "words", which in turn do the same things just one level higher.

Although no longer a commonly used, Forth has features that make it rewarding to study. The language has a wonderful minimalism that can be appreciated only with the study of example code.

Forth was a good compromise for building programs relatively easily, compared with assembler, that would use surprisely little memory and could be run surprisely fast. I built a system with a resident compiler, interpreter and Forth appliction code with concurrent threads all sitting in a few kilobytes of computer memory and running at about half the speed of the same program written in assembler.

According to Wikipedia, Forth is still used for such things as boot loaders, embedded systems and space applications (refs). There is an active implementation by the GNU Project.

In this chapter we'll build a small Forth compiler/interpreter system in Python and examine it in detail. Our implementation is very basic and small. Numbers (floats and ints) are the only data type, and I/O is only through the terminal.

Later on, we'll briefly look at the application mentioned above, a process control system for a newspaper mailroom. Sit back and enjoy the ride.

A word about quoting convention. If I use a word in all caps it is a Forth word. Our Forth is actually case insensitive, as you'll see, but traditionally Forth programs were in upper case. Words with mixed case are entities in the Python code. TOS is top-of-stack. Other words will be quoted when they stand for themselves.

Using the Data Stack

The "data stack" is the central feature, like the stove in your kitchen. And because we manipulate it directly, we have to compromise a little for the language. We have to get comfortable working in postfix.

Let's jump into an example. Click Here to access forth.py

```
$ python forth.py
Forth> 5 6 + 7 8 + * .
165
```

Here we are evaluating the product of two sums "(5+6)*(7+8)" which in normal infix notation requires that the sums be placed in parenthesis to ensure the multiplication follows the additions. Now, in a prefix notation, like that used in the Lisp language, one more pair of parans are needed "(* (+ 5 6) (+ 7 8))". But with Forth we use postfix notation and the need for parens simply disappears. Operators are simply placed where they are used in the computation.

(A side note. If, by chance, you know some Japanese then you are aware that thinking in postfix is not that unnatural)

We can see what is going on a little more clear by using the Forth word "dump" to display the data stack at any point in time. Let's go a little wild and show the data stack after each single operation.

```
Forth> 5 dump 6 dump + dump 7 dump 8 dump + dump * dump
ds =  [5]
ds =  [5, 6]
ds =  [11]
ds =  [11, 7]
ds =  [11, 7, 8]
ds =  [11, 15]
ds =  [165]
Forth>
```

Hopefully that will make it crystal clear.

Built in primitive words for data stack manipulation

Along with add, subtract, multiply and divide there are other basic builtin words that manipulate the stack as well.

- DUP duplicates the top of stack (TOS)
- SWAP interchanges the top two elements
- DROP pops the TOS discarding it
- "." pops and prints the TOS
- "=" pops the top two elements, compares them, and push 1 onto the stack if there are equal, and 0 if not

These can all be combined in clever ways. Here are two examples.

```
Forth> # Lets square and print the TOS
Forth> 25 dup * .
625
Forth> # Lets negate the TOS and print it
Forth> 42 0 swap - .
-42
Forth>
```

Here is the Python code for some of the basic runtime functions in forth.py. Our data stack is simply called "ds" and is a python list. We use the list "append" method to push items onto the stack and its "pop" method to get (and drop) the top element. Each basic word is a tiny function with two arguments. These functions do not use these arguments but we'll later see other runtime functions that will.

```
def rAdd (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a+b)
def rMul (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a*b)
def rSub (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a-b)
def rDiv (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a/b)
def rEq  (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(int(a==b))
def rGt  (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(int(a>b))
def rLt  (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(int(a<b))
def rSwap(cod,p) : a=ds.pop(); b=ds.pop(); ds.append(a); ds.append(b)
def rDup (cod,p) : ds.append(ds[-1])
def rDrop(cod,p) : ds.pop()
def rOver(cod,p) : ds.append(ds[-2])
```

```
    def rDump(cod,p) : print "ds = ", ds
    def rDot (cod,p) : print ds.pop()
```

Some of these could, of course, be done more efficiently but we'll opt for clarity. Notice the order of operations, especially with divide and subtract. Also, with the divide operator we are using the Python "/" so if at least one argument is a float, the result will be float. If both are integers, then integer division is done.

A reference to each of these functions is in the lookup table "rDict" which matches operators like "+" to their function (rAdd). These dictionary entries are straightforward.

Defining new words

Let's define a word NEGATE to replace the TOS with its negative. Every word definition starts with a ":" immediately followed by the word we want to define. Then everything following up to the closing ";" becomes the body of the definition. So let's define NEGATE and test it and then do the same for SQR.

```
    Forth> : negate 0 swap - ;
    Forth> 5 negate .
    -5
    Forth> : sqr dup * ;
    Forth> 6 sqr .
    36
    Forth>
```

Compiling Words to Pcode

The first step in compilation is lexical parsing, which in Forth is really simple. The "tokenize" function first strips comments, which in this little compiler is anything from "#" to the end of a line. Then it simply splits the text to a list of words, breaking on whitespace. Let's watch this from the Python prompt.

```
    >>> import forth
    >>> forth.tokenizeWords("5 6 + .  # this is a comment")
    >>> forth.words
    ['5', '6', '+', '.']
    >>>
```

When the compiler is not in the middle of defining a new word, or, as we'll see later, building a control structure, it operates in what is called "immediate mode". This means that a single word will be compiled and returned.

```
    >>> forth.compile()
    Forth> 5 6 + .
    [<function rPush at 0x9bdf5a4>, 5]
    >>> forth.compile()
    [<function rPush at 0x9bdf5a4>, 6]
    >>> forth.compile()
    [<function rAdd at 0x9bdf1ec>]
    >>> forth.compile()
    [<function rDot at 0x9bdf48c>]
    >>> forth.words
    []
    >>>
```

Notice how running the compile function automatically prompts us for input.

Compiled code comes in lists. A number, either a integer or float compiles to an "rPush" of the number onto the data stack. "+" and "." simply compile to their corresponding runtime functions. Let's look at the Python "compile" function.

```
def compile() :
    pcode = []; prompt = "Forth> "
    while 1 :
        word = getWord(prompt)  # get next word
        cAct = cDict.get(word)  # Is there a compile time action ?
        rAct = rDict.get(word)  # Is there a runtime action ?

        if cAct : cAct(pcode)   # run at compile time
        elif rAct :
            if type(rAct) == type([]) :
                pcode.append(rRun)     # Compiled word.
                pcode.append(word)     # for now do dynamic lookup
            else : pcode.append(rAct)  # push builtin for runtime
        else :
            # Number to be pushed onto ds at runtime
            pcode.append(rPush)
            try : pcode.append(int(word))
            except :
                try: pcode.append(float(word))
                except :
                    pcode[-1] = rRun     # Change rPush to rRun
                    pcode.append(word)   # Assume word will be defined
        if not cStack : return pcode
        prompt = "... "
```

Part of this may be a bit obscure. Basically, we get the next word using "getWord", which if it needs more input, will prompt for it. This "compile" function controls whether the prompt is "Forth> " or "... ". We'll see how that is used in a bit. If the word names a basic runtime action in "rDict", then it is translated to a call of that function. If the word can be made into an integer or float then "rPush" (also a builtin) becomes the translation followed by the actual number to be pushed onto the data stack. In immediate mode a single input word will be compiled and returned.

Next, let's compile a new word.

```
>>> forth.compile()
Forth> : negate 0 swap - ;
[]
```

No pcode is returned! But there is an interesting side effect

```
>>> forth.rDict['negate']
[<function rPush at 0x9fbe5a4>, 0, <function rSwap at 0x9fbe374>,
 <function rSub at 0x9fbe25c>]
>>>
```

We have a new entry in the rDict which can now be used just like a builtin

```
>>> forth.compile()
Forth> 6 negate
[<function rPush at 0x9fbe5a4>, 6]
>>> forth.compile()
[<function rRun at 0x9fbe56c>, 'negate']
>>>
```

Notice in the "compile" function above the check for cAct in cDict. This is looking for a compile time function. These functions are helper functions for the compilation process. Both ":" and ";" are compile time words. Let's look at their corresponding python functions

```
def cColon (pcode) :
    if cStack : fatal(": inside Control stack: %s" % cStack)
    label = getWord()
    cStack.append(("COLON",label))  # flag for following ";"
```

```
def cSemi (pcode) :
    if not cStack : fatal("No : for ; to match")
    code,label = cStack.pop()
    if code != "COLON" : fatal(": not balanced with ;")
    rDict[label] = pcode[:]       # Save word definition in rDict
    while pcode : pcode.pop()
```

You can see that an entry is pushed onto the cStack with the ":" and later popped by the ";". The ":" word gets a label from the input, and if it were not available you would be prompted for it with "...". This label is saved for the ";" word to connect it with the compiled code and make an entry in rDict. Once this happens the code is then erased. Now, since cStack is not empty during this time, compilation once started by ":" must proceed to the matching ";". Then we say that the compiler is running in the "deferred" mode.

So now let's look at other word groups that also use the cStack, forcing deferred compilation.

BEGIN ... UNTIL Control structure

The BEGIN and UNTIL words set up an iterative loop. UNTIL will pop the TOS and, if it is zero, will return control to the word following BEGIN. Here is an example

```
>>> import forth
>>> forth.main()
Forth> 5 begin dup . 1 - dup 0 = until
5
4
3
2
1
Forth>
```

We start by pushing 5 onto the stack, print it, subtract 1, and repeat until it is zero. Notice the use of the word DUP twice, needed to preserve our number as we are not only counting it down, but also printing it and checking for zero.

Both BEGIN and UNTIL are compile time words. Here are their corresponding Python functions

```
def cBegin (pcode) :
    cStack.append(("BEGIN",len(pcode)))  # flag for following UNTIL

def cUntil (pcode) :
    if not cStack : fatal("No BEGIN for UNTIL to match")
    code,slot = cStack.pop()
    if code != "BEGIN" : fatal("UNTIL preceded by %s (not BEGIN)" % code)
    pcode.append(rJz)
    pcode.append(slot)
```

BEGIN generates no code. It simply pushes the address (len(pcode)) of the next word to come onto the control stack and, thereby, puts the compiler into deferred mode. UNTIL checks that it has a matching BEGIN and generates an rJZ call (Jump if Zero) back to the address saved. At runtim rJz will pop the TOS and do the jump if it is zero.

Here is a nice "Forth-like" word definition for computing the factorial of the TOS. This code has been put into a file "fact1.4th"

```
# fact1.4th

: fact                              #  n --- n!  replace TOS with its factorial
  0 swap                            # place a zero below n
```

```
      begin dup 1 - dup  1 = until     # make stack like 0 n ... 4 3 2 1
      begin dump *  over 0 = until     # multiply till see the zero below answer
      swap drop ;                      # delete the zero
```

Notice the DUMP in the middle for debugging. Let's run this.

```
    >>> import forth
    >>> forth.main()
    Forth> @fact1.4th
    Forth> 5 fact .
    ds =  [0, 5, 4, 3, 2, 1]
    ds =  [0, 5, 4, 3, 2]
    ds =  [0, 5, 4, 6]
    ds =  [0, 5, 24]
    120
    Forth>
```

IF, ELSE, THEN Control Structure

The format of this control structure is

```
    condition IF true-clause THEN

      or

    condition IF true-clause ELSE false-clause THEN
```

Once again, if you are somewhat familiar with Japanese grammer, this ordering will not seem terribly unnatural. Let's look at the compile time helpers for these words.

```
    def cIf (pcode) :
        pcode.append(rJz)
        cStack.append(("IF",len(pcode)))  # flag for following Then or Else
        pcode.append(0)                   # slot to be filled in

    def cElse (pcode) :
        if not cStack : fatal("No IF for ELSE to match")
        code,slot = cStack.pop()
        if code != "IF" : fatal("ELSE preceded by %s (not IF)" % code)
        pcode.append(rJmp)
        cStack.append(("ELSE",len(pcode)))  # flag for following THEN
        pcode.append(0)                     # slot to be filled in
        pcode[slot] = len(pcode)            # close JZ for IF

    def cThen (pcode) :
        if not cStack : fatal("No IF for ELSE for THEN to match")
        code,slot = cStack.pop()
        if code not in ("IF","ELSE") : fatal("THEN preceded by %s (not IF or ELSE)" % code)
        pcode[slot] = len(pcode)             # close JZ for IF or JMP for ELSE
```

This should look familiar after studying the compile time code for BEGIN and UNTIL. IF will generate a rJz to the ELSE if there is one, or, if not, to the THEN. An ELSE will complete the jump for the IF and set up an unconditional jump to be completed by the THEN. THEN has to complete whichever jump is needed, from an IF or ELSE.

Here is a second factorial word definition. Instead of using iteration with BEGIN and UNTIL, we use recursion, a topic that will be discussed later in more depth. And rather than using a flag for establishing an end-point to the computation, we'll use our new IF, THEN team.

```
    # fact2.4th  Recursive factorial       # n --- n!
    : fact  dup 1 > if                      # if 1 (or 0) just leave on stack
              dup 1 - fact                  # next number down - get its factorial
      dump    * then                        # and mult - leavin ans on stack
      ;
```

Again, a DUMP has been stratigically placed to watch the action

```
    Forth> @fact2.4th
    Forth> 5 fact .
    ds =  [5, 4, 3, 2, 1]
    ds =  [5, 4, 3, 2]
    ds =  [5, 4, 6]
    ds =  [5, 24]
    120
    Forth>
```

Variables, Constants and the Heap

There comes a point when we need to store data outside of the data stack itself. We may want to more conveniently hold onto values independently of a changing data stack and we may want to store data in single or multi-dimensional arrays.

Forth systems set aside an area of memory for this purpose and contain a few builtins that are used to build words that, in turn, build other words. It's quite neat.

First, let's look at the builtins themselves. In our model the "heap" is simply a list of 20 integers. Obviously, it's not meant for real time.

```
    heap     = [0]*20      # The data heap
    heapNext =  0          # Next avail slot in heap

    def rCreate (pcode,p) :
        global heapNext, lastCreate
        lastCreate = label = getWord()      # match next word (input) to next heap address
        rDict[label] = [rPush, heapNext]    # when created word is run, pushes its address

    def rAllot (cod,p) :
        global heapNext
        heapNext += ds.pop()                # reserve n words for last create
```

Now, interestingly, both rCreate and rAllot are runtime words. When rCreate runs, the compiler is in immediate mode and the next word in the input (NOT in the pcode) will be the word being defined. The runtime action of the defined word will be to simply place its reserved heap address onto the data stack. ALLOT will actually reserve one or more words, advancing heapNext to be ready for a future CREATE. Let's look at an example. We'll use forth.main to both compile and execute our Forth code.

```
    >>> import forth
    >>> forth.main()
    Forth> create v1 1 allot
    Forth> create v2 3 allot
    Forth> create v3 1 allot
    Forth> dump
    ds =  []
    Forth> v1 v2 v3 dump
    ds =  [0, 1, 4]
    Forth>
```

Here we have created three new words V1, V2, and V3. Each has a corresponding heap address and when they are run each pushes its address. Notice the "space" between V2 and V3 due to the ALLOT used with the creation of V2.

And here is how we use these heap locations. Two builtins "@" and "!" fetch and set words in the heap. Let's play with this a bit

```
>>> import forth
>>> forth.main()
Forth> create v1 1 allot
Forth> create v2 3 allot
Forth> create v3 1 allot
Forth> ^D
>>> print forth.heapNext, forth.heap
5 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> forth.main()
Forth> 22 v2 !          # set v2[0] = 22
Forth> 23 v2 1 + !      # set v2[1] = 23
Forth> v2 @ .           # print v2[0]
22
Forth> ^D
>>> print forth.heapNext, forth.heap
5 [0, 22, 23, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> forth.rDict['v3']  # See definition of v3
[<function rPush at 0x8ba0454>, 4]
```

So, we created 2 simple variables (V1 and V3) and an array of 3 words V2. Dropping back into Python we can see the heap being modified and also a runtime definition for one of the variables.

Another useful builtin is "," which pops the TOS and pushes it onto the heap at the next free location. This can be used instead of ALLOT to initialize variables to zero.

```
>>> forth.main()
Forth> : varzero create 0 , ;
Forth> varzero xyz
Forth> xyz @ .
0
```

Finally, there is a builtin DOES> that works together with CREATE inside a definition. It takes all following words up to the closing ";" and appends them to the rDict entry for the word just created. First consider this definition and usage of a CONSTANT

```
Forth> : constant create , ;
Forth> 2009 constant thisYear
Forth> thisYear @ .
2009
```

That's fine, but there is nothing to prevent you from simply "!"ing thisYear to another value. It's the same as any variable. But if we instead do the following

```
Forth> : constant create , does> @ ;
Forth> 2009 constant thisYear
Forth> thisYear .
2009
Forth> ^D
>>> forth.rDict['thisyear']
[<function rPush at 0x9865454>, 3, <function rAt at 0x9865534>]
>>>
```

then you can see how a call to rAt has been attached to thisYear's runtime. There is no need to use an "@" to retrieve its value and "!" cannot be used to change its value.

With these builtins, it is not difficult to make arrays, including multidimensional ones. We can even make primitive structs (like C) assigning constants for field offsets.

Like C, however, there is not much memory protection. If an array reference is out of bounds, it will simply write into something else. Of course, in this model, this will stay within our heap but in a C adaptation of this code, even that would be lost.

We'll end this section with a final factorial program, this time using variables. The code looks more like a traditional language, albeit in postfix

```
# fact3.4th

: variable create 1 allot ;            #   ---     create var and init to TOS
  variable m
  variable answer

: fact                                 #  n --- n!  replace TOS with factorial
     m !                               # set m to TOS
  1 answer !                           # set answer = 1
  begin
    answer @ m @ dump * answer !       # set answer = answer * m
    m @ 1 - m !                        # set m = m-1
    m @ 0 = until                      # repeat until m == 0
  answer @                             # return answer
  ;

 15 fact .                             # compute 15! and print
```

And now, let's run it

```
Forth> @fact3.4th
ds =  [1, 15]
ds =  [15, 14]
ds =  [210, 13]
ds =  [2730, 12]
ds =  [32760, 11]
ds =  [360360, 10]
ds =  [3603600, 9]
ds =  [32432400, 8]
ds =  [259459200, 7]
ds =  [1816214400, 6]
ds =  [10897286400L, 5]
ds =  [54486432000L, 4]
ds =  [217945728000L, 3]
ds =  [653837184000L, 2]
ds =  [1307674368000L, 1]
1307674368000
Forth>
```

Since our model is built in Python, we inherit its nice automatic switch to long integers.

Notice that the variables "m" and "answer" are defined outside the "fact" definition. We don't have private local variables within a definition.

Other Issues

As mentioned earlier, Forth can be very efficient both with memory and CPU time. Consider the following bits of PDP-11 assembler code. It is a recreation of a little bit of our first Forth expression "5 6 +".

```
stack:
         ...       ;      more space here for the stack to grow
         6         ; <--- r4 stack pointer (stack is upside down)
         5
ds:      0         ; base of stack
```

```
          rPush     ; <--- r3 tracks the pcode thread
          5
          rPush
          6
          rAdd
          .
          .         ; thread continues ...


  rPush:  mov    (r3)+, -(r4)      ; ds.append[pcode[p]]; p += 1
          jmp   @(r3)+             ; return to thread

  rAdd:   add    (r4)+, (r4)       ; tmp=ds.pop(); ds[-1] += tmp
          jmp   @(r3)+             ; return to thread

  rDup:   mov    (r4),-(r4)        ; ds.append(ds[-1])
          jmp   @(r3)+             ; return to thread
```

This should look somewhat familiar from the assembler model in the previous chapter. We have the data stack on top, a bit of "threaded" code in the middle and 3 builtins. The threaded code (the name will be obvious in a minute) is essentially the same as our "pcode" array in the Python model. Machine register r3 is our "p" indexe to the next word in the pcode. The program counter, PC jumps between the builtins. The instruction "jmp @(r3)+" loads the program counter with the memory word indexed by r3 and then increments r3 to point at the next word. The program execution weaves through the threaded code out of one builtin (rPush) and into the next (rAdd). Register r4 is the ds index. On the PDP-11 the stack grew downward and lower machine addresses were actually higher on the stack. The instruction "mov (r3)+,-(r4)" pushes the next word in the thread (5 say) onto the data stack, first decrementing r4 to the next higher stack location.

Now if we were writing this in assembler we might do the following

```
  mov  #5, -(r4)        ; push 5 onto stack
  mov  #6, -(r4)        ; push 6 onto stack
  add  (r4)+,(r4)       ; add in stack
```

But if we add up the memory use and execution cycles, only the "jmp @(r3)+" instructions, the needle, if you will, that sews the code together are missing. These jumps constitute very little overhead.

In the early 1980's I developed a process control system for a newspaper mailroom that tracked bundles of newspapers on conveyor belts. Each bundle had a specific chute designation which would deliver it to a truck being loaded. We put together a small Forth system in less than 1000 lines of assembler. This system was concurrent having a special word so that one thread could yield control of the cpu to another. Like Forth itself, this application was divided into tiny pieces sewn back together. One pcode thread for example monitored keyboard input from one of the terminals. Another output messages to all the screens which included echoing input from either keyboard. Still other threads would handle a sensor on a belt or update a light display. Each thread had its own data stack but they shared variables on the heap and of course word definitions. There were no locking problems because the threads themselves yielded control only when safe to do so. Such a system was possible because throughput was only at human speed.

One final point. We used recursion in the second factorial example. This is unusual in Forth. Normally a word must be defined before being used in another definition. But in our compile function the last "except" clause allows us to build an rRun to an undefined word with the assumption that it will be defined before it is actually used. But this in turn leads to another issue. Our rRun runs a word dynamically, that is, it looks up the definition in rDict just before running it. Most Forths would not do that. It's expensive when the computation for most words is usually so small. So rather than following a pcode entry "rRun" with the name of a word, it would be reference to the words pcode and the dictionary lookup is avoided. This also has an interesting implication. If you redefine a word that has been used in the definition of other words, those other words do not change their behaviour. They are still locked to the old code. The programmer might well find this unexpected.

-