Read-Eval-Print-λove

v003 - Forth Thinking

Michael Fogus

This book is for sale at http://leanpub.com/readevalprintlove003

This version was published on 2016-03-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

world hello

The Forth programming language was created by Charles (Chuck) Moore as the basis for a personal programming environment. After working on this system for some time in isolation, Moore decided to extract a Forth implementation for use in the National Radio Astronomy Observatory's 11-meter radio telescope in 1971. Imagine this for a moment if you please. Chuck Moore, some time in 1968 found that the computing environments of the day were insufficient for his needs and decided to create his own from whole-cloth. Over the next 2-3 years Moore developed a computing environment suited to his philosophy of programmatic style. A crucial components of this system was the programming language forming its underpinnings: Forth.

By modern standards, Forth is an odd duck. First, the language is stack-centric, meaning that every operation in the language deals either implicitly or explicitly with a stack-based programming metaphor. This is decidedly different from a programming language like C that may use as an implementation details an underlying stack to pass function activations. In Forth, a program consisting of a single token can demonstrate how its stack is central in its programming model:

```
1  9
```

The Forth program above is interesting in that it illustrates the conceptual underpinnings of the language. In most modern programming languages, a program containing the number `9` would be quite pointless, but in Forth such a program would cause an effect in the Forth environment itself. That is, the number `9` in Forth isn't just a constant integer, instead it's a command to the Forth environment that states "push the integer `9` onto the data stack." The data stack in Forth is a global scratchpad in Forth environments used by operations to retrieve and store the results of functions.

A larger Forth program is shown below:

```
1  36 9 /
```

In addition to the commands to push `36` and `9` onto the stack, there's a new command called `/` that states: "pop something off of the stack and make it the denominator and then pop another thing off and make that the numerator and then push the resulting quotient onto the stack." At the end of this program you'll not be surprised to learn that the number `4` is the only element on the data stack.

Before I go any further I'd like to talk a little bit about the word "concatenative." Very often when discussing Forth and languages of its ilk (the pedigree being Factor[1], Joy[2], Open Firmware[3], and

[1] http://factorcode.org/

[2] http://c2.com/cgi/wiki?JoyLanguage

[3] http://www.firmworks.com/

PostScript[4]) you'll see this term bandied about, but what does it really mean? From a simplistic perspective, the term concatenative means "programs are built by laying code words side by side," but again this is nearly meaningless. Instead, let me create a very simple little Clojure function that might help clarify the matter:

```
(defn postfix [& words]
  (reduce #(if (fn? %2)
             (let [[l r & m] %]
                       (cons (%2 r l) m))
                       (cons %2 %))
                  []
                  words))
```

The `postfix` function is fairly straight-forward. It takes a number of arguments (`words`) that represent either functions or non-functions. A sample call looks as follows:

```
(postfix 5 1 2 + 4 * + 3 -)
```

It then walks the `words` and when it finds a function it calls it with the top-2 elements of the "stack" (the `words` list) and plops the result back in place, otherwise it just plops the `word`. Eventually, what comes out of `postfix` is something that represents a stack, a la:

```
(postfix 5 1 2 + 4 * + 3 -)
;;=>  (14)

(postfix 3 2 1 + *)
;;=> (9)

(postfix 1 2 3 * +)
;;=> (7)
```

Now imagine that the vector `[5 4 *]` represents a valid program:

```
(apply postfix [5 4 *])
;;=> (20)
```

I could literally *concatenate* another valid program onto this vector in order to create a new valid program:

[4] http://www.tailrecursive.org/postscript/postscript.html

```
(apply postfix (concat [5 4 *] [30 10 +]))
;;=> (40 20)
```

Likewise, I could concatenate an operator into that result to create yet another valid program:

```
(apply postfix
  (-> [5 4 *]
      (concat [30 10 +])
      (concat [+])))
;;=> (60)
```

Ad infinitum.

As shown, programs built with concatenative languages are composed via the act of "laying" operators, or even whole programs, side by side. Additionally, like in natural language, Forth words can directly replace longer phrases. Observe:

```
5 dup *
```

When the program above is run, the data stack will look as follows:

```
|  25  |
|      |
+------+
```

That is, the only element on the stack will be the number 25 because the program `5 dup *` means "duplicate the thing on the top of the stack and push the duplicate on top of it, then take two things off of the stack (two 5s) and multiply them together, pushing the result back onto the stack." The program fragment `dup *` is therefore equivalent to the act of squaring the number on the top of the data stack. Indeed, I could use those two words to define a new word, as shown below:

```
: sq dup * ;
```

I'll talk more about the `:` operator later, but for now you only need to understand that it creates a new word named `sq` consisting of the two words `dup *`. Now, I can directly replace the original longer phrase `dup *` with `sq` to keep the original behavior:

```
5 sq
```

This is a common work-flow in Forth-like languages:

- Write some code to perform a task
- Identify some fragment that might be generally useful
- Extract it and give it a name
- Replace the relevant bits with the new word

There is nothing special about this work-flow except that it's indicative of a philosophical underpinning of Forth-like languages driving code toward a highly-layered implementation. I'll dive into this angle later.

In this installment of Read-Eval-Print-Love I'll discuss Forth by first dissecting a small Ruby implementation called "Russ-Forth"[5], which is a simplified Forth (e.g. no return stack). Once the runtime for Russ-Forth is in place I'll implement the bare bones of a core library and talk about stack-shufflers, combinators, and the like. I'll then talk about the ideas in the amazing book "Thinking Forth" by Leo Brodie which describe an interesting bottom-up approach to system design.

While Forth is an interesting programming language in its own right, the language itself is a philosophical statement by Chuck Moore (Brodie 1984) about how programs should be constructed. Of course, I'll have fun with a tiny Forth-like implementation, but the point of this installment is to establish a context for describing the Forth philosophy. I'm personally of the opinion that far more important than the programming language itself, the *idea of Forth* is key.

[5]The name is an homage to my friend and colleague Russ Olsen, who posted the original code to me as a comment on one of my GoodReads reviews. Thanks Russ!

Russ Forth: a simple Forth in Ruby

This wouldn't be an issue of Read-Eval-Print-Love without an implementation of a programming language, so here we are again. This time I'm going to implement a little version of Forth that looks a little bit like a Forth that you might find out in the wild,[6] but that is greatly simplified.

The implementation of a small Forth interpreter like Russ Forth (RForth) is fairly small as you'll see in the next few pages. Aside from the core functions, RForth consists only of a lexicon to store *word* (functions) definitions:

```
1 require_relative 'lexicon'
```

A compiler to convert those word definitions to core constructs – in this case, Ruby constructs:

```
1 require_relative 'compiler'
```

And a token reader to handle the raw text input:

```
1 require_relative 'reader'
```

While these three pieces will provide the core functionality of the Forth scanning and execution, RForth needs more than that. Indeed, while not comprehensive, I will implement a small set of core functions found in many concatenative languages:

```
1 require_relative 'shufflers'
2 require_relative 'stack_ops'
3 require_relative 'io'
4 require_relative 'math'
5 require_relative 'comparators'
```

I'll talk more about these core words later. For now let me walk through the implementation of the main RForth driver class.

[6] If you're interested in actually using a Forth for personal or professional use, then there are far better options than Russ Forth. At the end of this installment I'll list a few.

Initialization

Any good Ruby program starts with a `class` definition, and this one is no different: [7]

The `RussForth` class will control the text scanning, compilation, and execution interpreter pipeline. On initialization the class is instantiated with the input and output channels, that are set to standard in and out (console I/O) by default.

```
1       @lexicon = Lexicon.new
2       @reader = Reader.new( @s_in )
3       @compiler = Compiler.new
```

As mentioned before, three key components of the system are implemented as a set of three classes: `Lexicon`, `Reader`, and `Compiler`. However, the core metaphor of the system is implemented as a simple Ruby array:

```
1       @stack = []
```

However, rather than being used simply as a array, the `@stack` property will be managed as, well, a stack. You see, Forth is a stack-centric concatenative programming language.[8] All arguments into and results from words pass from one to the other via the system stack. Since I talked about this in the introduction I won't belabor the point here, but I might talk more about it in situ whenever appropriate.

```
1       build_lexicon
2     end
```

The final stage of the initialization process is the population of the lexicon with core words. I'll discussion this more later.

Evaluation

The RForth evaluation model is very simple. Indeed, much of the simplicity is owed to the fact that it farms off much of the execution logic to Ruby itself. That is, after the read and compilation phases RForth words are left as Ruby `Proc` instances awaiting execution which are stored in the `Lexicon`. Programmatically this looks as follows:

[7] Save for perhaps the "good."
class Russforth def initialize(s_in = $stdin, s_out = $stdout) @s_in = s_in @s_out = s_out

[8] Most concatenative programming languages use a stack as the core metaphor for computation, but it's not necessarily the case that concatenative languages *must* do so. Indeed, if I ever get around to writing an installment of Read-Eval-Print-Love about concatenative languages in general then I'll build a little interpreter for a language that does not use a stack at all. That's a task for another day however.

```
1   def evaluate( word )
2     entry = resolve_word(word)
3
4     if entry
5       entry[:block].call
6     else
7       @s_out.puts "#{word} ??"
8     end
9   end
```

As you might have noticed, the `entry` retrieved is actually a hash containing a `Proc`. The reason for this level of indirection is because there's a little extra information needed pertaining to the word that's stored in the hash returned from `resolve_word`. The implementation of `resolve_word` is as follows:

```
1   def resolve_word( word )
2     return @lexicon[word] if @lexicon[word]
```

First of all, if a word is in the `@lexicon` then `resolve_word` simply returns it as is. However, if the word is not found then RForth needs to perform some careful logic:

```
1     if word[0,1] == ":"
2       x = word[1..-1].to_sym
```

Since this is a Ruby-based interpreter, I thought it would be fun to handle Ruby symbols. Therefore, the first check is to see if the read word starts with a colon. If it does, then the word is just the symbol itself – there's a good reason for this, as I'll talk about in a moment. But first, if the word is not a symbol then it might be a number:

```
1     else
2       x = to_number(word)
3     end
```

Without going into too much detail about `to_number` (I'll inflict that on you in a moment) the point is that should the word in fact be a number then it too is simply the number itself. So in the cases where a word resolves to a symbol or a number, the RForth interpreter has to handle them in an interesting way:

```
    if x
      block = proc { @stack << x }
      return { :name => word, :block => block, :immediate => false }
    end
```

That is, if the word resolved to either a number or a symbol then RForth builds a Ruby `Proc` that does nothing but push them onto the top of the stack. You'll sometimes hear that everything in Forth (and concatenative languages in general) is a function (word) including scalars like numbers and the like. Indeed, the rationale is sound given that it's often important to perform calculations involving numeric constants and symbols and since that is the case, it makes sense to leverage the paradigm of the language itself to add such constructs. While not every concatenative language utilizes this trick, instead preferring more performant options, I chose to because I'm drawn to the purity of the idea.

```
    nil
  end
```

Finally, if the word was none of a process in the lexicon, symbol, nor number then of course it was nothing at all. Real quick, allow me to show the abominable `to_number` implementation below:

```
  def to_number( word )
    begin
      return Integer( word )
    rescue
      puts $!
    end
    begin
      return Float( word )
    rescue
      puts $!
    end
    nil
  end
```

Rather than going into detail about this method, I'd prefer to turn a blind eye and pretend that it never happened. Instead, allow me to say that the above is all that there is to the evaluation model of a simple Forth like RForth. However, while it might make sense that numbers and symbols are self-pushing words, there are more complicated words comprising the RForth run-time. First, there are internal words implemented as Ruby methods and while I'll get to those eventually, I'm more interested in user-defined words supported by the language itself. Of course, the Devil's in the details and so in the next few sections I'll dig into the (sometimes gnarly) details.

Reading and defining words

I glossed over the action of processing user-defined words, so let me take a few paragraphs to go back and discuss it a bit. Rather than dig into the details of tokenization and character-by-character lexing, I'll keep things at a higher level.

```
1   def read_and_define_word
2     name = @reader.read_word
3     words = []
```

Speaking of char-by-char lexing, the bulk of that task is farmed out to the `Reader` class which I'll discuss later. It's assumed, that `Reader#read_word` "knows" what it's doing and will return a token corresponding to the name of the word. If you recall, the `read_and_define_word` was initiated when the `resolve_word` method encounter a colon. The colon started the user-defined word and a semicolon ends it – in between are just regular RForth words:

```
1     while (word = @reader.read_word)
2       break if word == ';'
3       words << word
4     end
```

Once the constituent words comprising the body of the user-defined word are read, they are then compiled into a Ruby `Proc`:

```
1     p = @compiler.compile_words(self, *words)
2     @lexicon.define_word(name, &p)
3   end
```

A method that's closely related to `read_and_define_word` is shown below:

```
1   def read_quotation
2     words = []
3     while (word = @reader.read_word)
4       break if word == ']'
5       words << word
6     end
7
8     @stack << @compiler.compile_words(self, *words)
9   end
```

You'll notice that the basic structure of `read_quotation` is almost exactly like that of `read_and_define_word` except for what happens to the returned `Proc` from `Compiler#compile_words`. That is, rather than define the word in the `@lexicon`, it's stored directly onto the `@stack` itself. For the Lisp programmers in the audience you might realize that this is the Forth analogy to a lambda or anonymous function. The utility for these anonymous words, called "quotations" in Forth, will be shown later when I talk about RForth combinators.

I'll also talk a little about the compilation step later, but for now let me move on to the final parts of the `RForth` class: driving and seeding.

Run-time

I've already shown the basic parts of the system, but it's the `run` method that ties it all together:

```
1   def run
2     until $stdin.eof?
3       @s_out.flush
4       word = @reader.read_word
5       evaluate word
6     end
7   end
```

It couldn't possibly be easier. Read, Eval, Loop. However, it gets a little more interesting when a language proper is built on top of this simple frame. Commercial-grade Forth systems typically come with hundreds of general-purpose and domain-specific[9] words out of the box. However I'm only going to talk about a handful of word classes for the purpose of illustration, including:

I'll dive into each of these soon, but for now all that I want to say is that the words are defined as Ruby modules and imported to the RForth lexicon at launch time:

```
1   def build_lexicon
2     @lexicon.import_words_from Verbs::Shufflers, self
3     @lexicon.import_words_from Verbs::StackOps, self
4     @lexicon.import_words_from Verbs::Io, self
5     @lexicon.import_words_from Math::Arithmetic, self
6     @lexicon.import_words_from Verbs::Comparators, self
```

Each of the built-in words are implemented as methods contained in modules and aliased for use in the system:

[9] I'll dive deeper into Forth and domain specific programming later in this installment.
include Verbs::Shufflers include Verbs::StackOps include Math::Arithmetic include Verbs::Comparators include Verbs::Io

```
    @lexicon.alias_word('?dup', 'qdup')
    @lexicon.alias_word('+', 'plus')
    @lexicon.alias_word('*', 'mult')
    @lexicon.alias_word('-', 'subtract')
    @lexicon.alias_word('/', 'divide')

    @lexicon.alias_word('=', 'eq')
    @lexicon.alias_word('<>', 'noteq')
    @lexicon.alias_word('>', 'gt')
    @lexicon.alias_word('<', 'lt')

    @lexicon.alias_word('.', 'dot')
    @lexicon.alias_word('.S', 'dot_s')

    @lexicon.define_word(':') { read_and_define_word }
        @lexicon.define_word('[') { read_quotation }
    @lexicon.define_word('bye') { exit }
```

However, there is a special kind of word supported by most Forth implementations called *immediate words* and I'll talk about those later as well.

```
    @lexicon.define_immediate_word( '\\' ) { @s_in.readline }
  end
end

Russforth.new.run
```

The basic structure of the RForth system is fairly straight-forward as most of the capability is farmed out to supporting classes, each of which I'll discuss presently in turn.

Lexicon

The RForth `Lexicon` class just implements a fancy hash-map: [10]

```
class Lexicon
  def initialize( &block )
    @entries = {}
  end
```

Since the `Lexicon` is a map then it should behave like one:[11]

[10] Traditionally, Forth lexicons are implemented as linked lists to avoid overwriting previous word implementations.

[11] I've always liked Ruby's capability to override the `[]` array look-up as it's allowed me to use Ruby think in terms of associative data rather than merely object-centric.

```
1   def []( name )
2     @entries[name]
3   end
```

What's stored in the map is a k/v pair of name/descriptor. The descriptor is yet another map that just describes the word and provides its implementation:

```
1   def define_word( name, &block )
2     @entries[name] = { :name => name, :block => block, :immediate => false }
3     self
4   end
```

Immediate words, or words that are executed during the read phase, differ from regular words only by a flag:

```
1   def define_immediate_word( name, &block )
2     @entries[name] = { :name => name, :block => block, :immediate => true }
3     self
4   end
```

Words can have aliases, which are defined in terms of previously stored words in the `Lexicon`. The utility of this will become apparent when I talk about the `import_words_from` method below.

```
1   def alias_word( name, old_name )
2     entry = self[old_name]
3     raise "No such word #{old_name}" unless entry
4     new_entry = entry.dup
5     new_entry[:name] = name
6     @entries[name] = entry
7   end
```

Nothing about the `Lexicon` has been terribly interesting so far (which is why I glossed over the details). However, its implementation is slightly more interesting in the way that it imports base words from Ruby modules. That is, the core words in RForth are implemented within Ruby modules and actively imported at start time to load the core `Lexicon` instance, as shown below:

```
  def import_words_from( mod, context )
    mod.public_instance_methods(false).each do |meth|
      method_closure = context.method(meth.to_sym)
      define_word( meth.to_s, &method_closure )
    end
  end
end
```

As shown, the public methods defined in a passed module are closed over individually and stored in the `Lexicon` using their defined names. Perhaps now you see the utility of the `alias_word` method defined above. That is, the limitations of Ruby method naming shouldn't bleed into the RForth word name definitions and so while it's straight-forward to import a module method directly, the presence of `alias_word` allows me to fix up the name to something more appropriate to RForth. You'll see this in action later, but for now I'll talk about the class responsible for the read phase of the RForth implementation.

Reader

Forth languages, in general, do not have a read phase like you'd find in Lisp languages. Instead, the term `Reader` is meant to encapsulate the logic for an RForth phase pertaining to the reading of words from an I/O stream. This is more like a classical language scanner except that it performs no level of token classification. Instead, since everything in RForth is a word then everything scanned is assumed to be a word. This is perhaps a naive implementation, but it works for my purposes.

```
class Reader
  def initialize( in )
    @input = in
  end
```

The `Reader` class is initialized with an input stream. There's nothing surprising about that fact perhaps. As shown in the introduction, RForth words are always separated by white-space, (i.e. tab, space, newline, etc.) so the Reader needs a way to identify when it encounters some:

```
  def is_space?( ch )
    /\W/ =~ ch.chr
  end
```

Regular expressions to the rescue![12]

This is all fine and good, but the real meat of the word identification is shown below:

[12] I'll avoid the temptation to add a certain famous quote about regular expressions. Instead, I will say that pound for pound regexes are amongst the most powerful programming abstractions going. That said, the density of information packed into a regexes leads me to minimal their footprint in my own programs to minimize the occurrences of cursing for my 2-weeks-later-self.

```
  def read_word
    result = nil
    ch = nil
    until @input.eof?
      ch = @input.readchar
      if result and is_space?(ch)
        break
      elsif result.nil?
        result = ch
      else
        result << ch
      end
    end
```

That little nasty bit of code is (unfortunately) obfuscating a simple bit of logic that can be stated simply as "append a bunch of characters to a word until a space or `nil` happens along." Once one of those separators are encountered then RForth just assumes that it's found a word and just returns it:

```
    return result if result
    nil
  end
end
```

And that's the entirety of the `Reader` class. This could be made more robust at the expense of clarity, but I decided to keep it simple for now. Frankly, only a few odd people (like myself) care about the minute details of lexical scanning, so by assuming that every token was potentially a word I was able to forego a whole swath of complexity.

I'll follow this same complexity conservation pattern in the next section in the implementation of the RForth compiler.

Compiler

I agonized long and hard over whether I would refer to this class as a compiler or a more terrible word (perhaps one starting with a 'T'), but in the end my biases show through. The `Compiler` class is more to stick to the layered phase separation of RForth than a traditional compiler. Indeed, the implementation is such that RForth words comprising a program are aggregated into Ruby `Proc` instances for execution. This is not problematic until you realize that the `Compiler` translates RForth programs into sequences of `Proc` calls which then call other `Proc` instances that then bottom out as Ruby module methods. That is, the compilation phase for RForth is very tightly coupled to the RForth run-time environment. In other words, the `Compiler` class is simply responsible for translating RForth words into sequences of pre-existing Ruby method calls. That said, it still might be interesting to explore a little.

```
class Compiler
  def compile_words( context, *words )
    actions = []
```

The compiler will take an array of words and transform them, collecting the results into an array of `Block` instances.

```
    words.each do |word|
      entry = context.resolve_word( word )
      raise "no such word: #{word}" unless entry
```

First, each word is expected to exist in the given context (probably a `RussForth` instance) and throw an error if it can't be resolved.

```
      if entry[:immediate]
        entry[:block].call
```

If the resolved word happens to be an immediate word then it's executed… well… immediately. Immediate words are the Forth way to define compile-time effects and I'll talk more about that later.

```
      else
        actions << entry[:block]
      end
    end
```

However, if the resolved word is a normal word then its block attribute (the part that performs the action) is stored for later use:

```
    proc do
      actions.each {|b| b.call}
    end
  end
end
```

That is, the result of the `compile_words` method is a Ruby `Proc` that closes over the `actions` and executes them each in turn at some point in the future. This is the whole mechanism behind how user-defined words are "compiled" into Ruby `Procs`.

Conclusion

The conceptual framework for a Forth-like language is surprisingly small. With even a moderately expressive language one can create a Forth-like in very few lines of code, required very little conceptual bulk. That said, even though I've shown the guts of RForth, I've done very little to describe the kinds of programs and patterns that fall out of a Forth-like language. The rest of this installment will be devoted to just that topic with a particular eye towards demonstrating how such a simple language can lead to deep implications on how programs are reasoned about and constructed.