CPL

(Dr)Racket

2016

November 2016

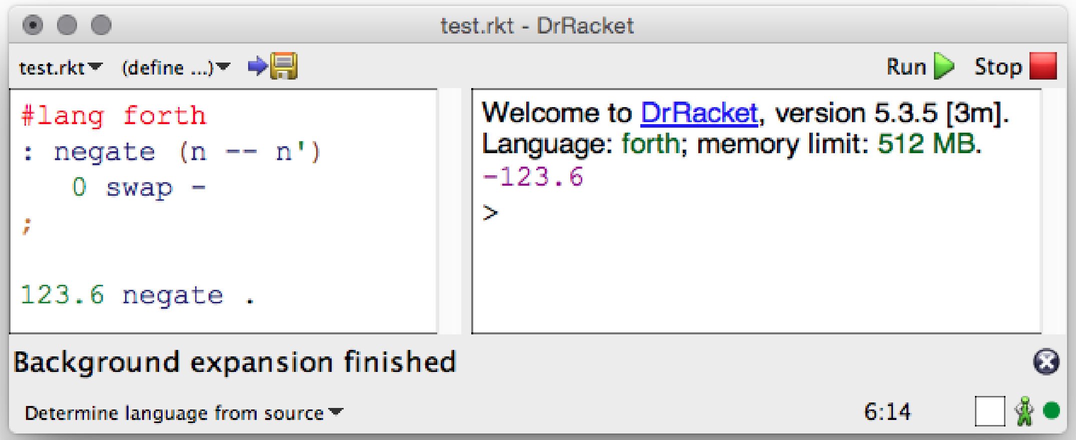
Abstract

This document is a step-by-step tutorial on developing a basic Forth interpreter in Racket. It is an integral part of the course on “Comparative Programming Languages" of prof. Frank Piessens.

Contents

Exercise Session

In this exercise session, we will use your basic knowledge about Racket and turn them to use while exploiting Racket's programming language lab features. Your task today is to *implement a simple Forth interpreter* in Racket, using some more advanced and typical Racket aspects. In the end, your interpreter will be able to execute programs (see a later section for the semantics) like this one:



```
#lang forth
: negate (n -- n')
   0 swap -
;

123.6 negate .
```

```
Welcome to DrRacket, version 5.3.5 [3m].
Language: forth; memory limit: 512 MB.
-123.6
>
```

Background expansion finished

Figure 1:

The *first* part of this document explains the necessary steps before development: what to download and how to set up your environment. The *second* part starts with a description of the Forth language itself. Next, it describes, step-by-step, how to create your own Forth interpreter (indicated by **Task**). Please, first read this document carefully, as it contains many hints that will save you time in the end. Only start programming when you encounter **Task**s.

This assignment will not be graded and as such doesn't require you to hand something in. After the session, a example solution will be made available on Toledo.

Background

Forth is an imperative, stack-based computer programming language without type checking.

Although not as popular as other programming systems, Forth has enough support to keep several language vendors and contractors in business. According to Wikipedia, Forth is currently used in boot loaders such as Open Firmware, space applications, and other embedded systems.[1]

Forth has a set of basic commands or subroutines (called words). The programmer can construct, on top of the basic commands, new commands that are immediatly available within the language. The accessible parameter stack can be used to put values on and to store the result of calculations. Therefor, global variables are (almost) never used.

Setup

For this exercise session you will only need a working installation of DrRacket and a web browser to browse the documentation on http://docs.racket-lang.org/.

Forth Skeleton

Download the `forth.tar.gz` file from Toledo and unpack it in your home directory `/home/<studentnumber>` with the command : `tar zxf forth.tar.gz`.

Make sure that the `/home/<studentnumber/forth` directory contains all the necessary files.

Task: Inform the current Racket installation that you are creating a new library. Open a console window and invoke `raco link --name forth /home/<studentnumber/forth`.

Now you are ready to start developing!

Forth Internals

In what follows, we will go through some fundamentals of the Forth language. We will cover some concepts that you need to know and the different ways how to interact with them. Read them carefully as they are fundamental for solving this exercise session.

Stack

The "data stack", or stack, is a central feature in Forth. Programs will directly manipulate this stack.

Each number the interpreter receives, will be pushed onto this stack. Our Forth implementation will need some built-in functionality for stack manipulation:

1. `dup` duplicates the top element of the stack
2. `swap` interchanges the top two elements
3. `drop` removes and discards the top element

[1]Wikipedia: Forth (programming language)

4. `.` or `pp` pops and prints the top element
5. `=` pops the top two elements, compares them, and push 1 onto the stack if there are equal, and 0 otherwise
6. `dump` prints the elements of the stack from top to bottom to the screen, left to right

See the first task in this document.

Words

Forth has the ability to call "snippets of computation", referred to as "words" that manipulate the data stack directly or get strung together to build (compile) new "words", which in turn do the same things just one level higher. Most words are specified in terms of their effect on the stack. Typically, parameters are placed on the top of the stack before the word executes. After execution, the parameters have been erased and replaced with any return values.

Our Forth implementation will have some built-in words and will allow a programmer to define their owns. Words in Forth exists out of ASCII characters plus space, which separates individual words. Forth comes with an implementation of the typical mathematical operations like `add`, `subtract`, `multiply` and `divide` :

```
5 4 dump + dump .
> [ 4 5 ]
> [ 9 ]
> 9
```

New words can be added to the language on the fly. Every new word definition starts with a `:` immediately followed by the implementation of the word you want to define. A definition is closed with a `;`. A possible definition of `negate` could look like:

```
: negate 0 swap - ;
5 negate .
> -5
```

Comments

Comments in Forth can be written between `( ... )`.

Typically, right after the name in word definition, one puts the high-level behavior of the word:

```
: negate (n -- n')
    0 swap -
;
```

A common thing for Forth programmers is to use the `(n -- n')` comment to indicate that the word consumes an `n` and produces another, possibly different, `n'`.

Limitations

We introduced some limitations for this exercise session to simplify and speed things up:

- Predefined words, like the mathematical operators can't be redefined.
- Literals can't be used on the data stack.
- There are no explicit I/O words nor words to explicitly work with memory.
- We will leave error handling as future work. For now, you can expect a programmer to behave for good.

Forth Machine

The first implementation step is to design the internals of our Forth machine, our so-called *semantics*. The model of our Forth machine is rather basic. Working with a stack, is an essential feature of our Forth machine. Therefor, we will need a state for our machine and we will need a data structure for the stack and a dictionary (implemented as a hash table) for words definitions.

Our basic implementation looks like this:

```
#lang racket
(provide (all-defined-out))

(define-struct state (stack words) #:mutable)

(define (new-state)
  (make-state '() (make-hash)))

; your implementation of the semantics
```

The second line makes sure that all defined functionality will be exported. The following example shows how to use `define-struct`. Use the online Racket manual to learn more its behaviour.

```
(define-struct point (x y) #:mutable)
(define p (make-point 1 2))
(set-point-y! p 3)
(cons (point-x p) (point-y p))
> '(1 . 3)
```

Next you need to make sure that you can interact with the state. These interactions will form the basis for our machine.

Task (semantics.rkt): Implement `(push-stack val a-state)`, `(pop-stack a-state)` and `(dump-stack a-state)` to interact with the stack. The `push-stack` command pushes a value on top of the stack, `pop-stack` removes the top element of the stack and `dump-stack` prints out each element of the stack. You can always check the file `tests.rkt` (see next Section for more info) to gain extra insights in the expected behavior.

Task (semantics.rkt): Implement `(get-word name a-state)` and `(set-word name body a-state)` to interact with the `words` part of the `state` struct. The `get-word` command searches in the `words` hash table, while the `set-word` command (over)writes an entry. If you are not familiar with hash tables in Racket, look for Section 3.10 in the online Racket Guide. Again, if in doubt, check the `tests.rkt` file to gain extra insights.

After this task, we have a basic version of the semantics of our Forth machine. In the next section, we will learn how we can test our code.

Testing

RackUnit is a unit-testing framework for Racket. There are three basic concepts in RackUnit:

1. A check is the basic unit of a test. As the name suggests, it checks some condition is true.
2. A test case is a group of checks that form one conceptual unit. If any check within the case fails, the entire case fails.
3. A test suite is a group of test cases and test suites that has a name.

Make sure that you write enough tests to verify that your code works as expected. As you will continuously build upon code, it is of utterly importance that your code works as expected.

Task (tests.rkt) While testing, make sure that you don't include files that you haven't modified yet or that you don't try to test functionality that hasn't been developed yet. Most likely, you will have to rely on commenting (it is available throught DrRacket or by starting a line with `;`) You can add as many new use cases as you want. Complete this task by making sure that your basic Forth semantics work as expected.

Language

A first step in our journey for a Forth interpreter, is to allow programmers to write Forth using s-expressions. Essentially, this comes down to writing programs as abstract syntax trees (AST) of Forth.

The expressions in these ASTs will rely use the semantics you've coded earlier and will define syntax rules (via `define-syntax-rule`) for handling all the built-in words and basic interactions with our machine.

We want every Forth program to run independently from others. This means that each Forth program will run on its own Forth machine and thus that it needs a fresh state. We will exploit the built-in module system from Racket, by overwriting `#%module-begin`, to embed code for a new, fresh state:

```
(module test "./language.rkt"
(#%plain-module-begin
    (parameterize ((current-state (new-state))
    (word "negate" (num 0) (swap) (min))
    (num 5)
    (call "negate")
    (pp))))
```

The state of the Forth machine can be accessed via `current-state` which is some sort of a stateful global variable.

Task (language.rkt): Define and implement the syntax rules for `(dump)`, `(num v)`, `(word name body ...)`, `(plus)`, `(min)`, `(mul)`, `(div)`, `(dup)`, `(swap)`, `(drop)`, `(pp)`[2], `(=)`, by carefully interacting with the Forth machine (via `current-state`) you have implemented earlier. To save the body of a word definition, you can use the list notation `'(body ...)`. This list can be the empty list if no word body is given. The `(num v)` construct represents an integer in the Forth AST. The `negate` example from the beginning of this document, has the following AST representation:

```
(word "negate" (num 0) (swap) (min))
(num 5) (call "negate") (pp)
```

After finishing this task, you should be able to run complete Forth programs written with s-expressions in the REPL!

Task (tests.rkt): Carefully test your implementation of the syntax rules. For example, to test your syntax rules, you can use `expand` to see to what they expand:

```
> (syntax->datum (expand '(num 1)))
'(#%app push-stack '1 (#%app current-state))
```

The `#%app` indicates that this is a function application. You can use your s-expressions in the 'Interactions' window, add them at the end of your file or open a new file (saved within the `forth/` directory) and enter

```
#lang s-exp "language.rkt"

(num 1)(num 2)(dump)(plus)(dump)
```

This will DrRacket to interpret the syntax as s-expressions and to treat it with the language module `language.rkt` that you just wrote up.

Calling words

Task (language.rkt): Implement `(call name)`. For this to work, you'll need to rely on the built-in dynamic evaluation mechanism, `eval`, with definition `(eval top-level-form namespace)`. The `top-level-form` must be the word you want to execute, and `namespace`

[2]You can use `(pp)` as a replacement for the `.`

must be the previously pre-defined `ns`. This makes sure that the dynamic evaluation happens within our language definition.

Forth Interpreter

To allow Forth programmers to write Forth in the surface syntax, we provide you with a simple parser that transforms real Forth code into an AST of s-expressions you have defined earlier. The parser goes from Forth syntax to the s-expressions as defined in your `language.rkt`.

You now have the language and a parser. Let's tie them together! This part is fairly straightforward, because you have the two pieces in hand:

1. A parser in `parser.rkt` for the surface syntax that produces ASTs
2. A module language in `language.rkt` that provides the meaning for those ASTs.

To combine these two pieces together, you want to define a reader that associates the two. When Racket encounters a `#lang` line of the form:

```
#lang forth
```

it will look for a reader module in `lang/reader.rkt` within the `forth` module, and use it to parse the file.

The reader language for `#lang` is similar to `s-exp`, in that it acts as a kind of meta-language. Whereas `s-exp` lets a programmer specify a module language at the expander layer of parsing, reader lets a programmer specify a language at the reader level. For `#lang`, the `read` and `read-syntax` functions must produce a module form that is based on the rest of the input file for the module. Racket provides a helper module called `syntax/module-reader` to handle most of the dirty work with modules.

Task (lang/reader.rkt): Implement the `(my-read-syntax src in)` function.

Task Run the example `test.rkt` and verify your result.