Functional Programming From First Principles

uocsclub

November 4, 2021

Contents

1 Scheme

Today we're learning functional programming from first principles, and the vehicle through which we're doing it is Scheme. In the process, we'll be discovering a whole other side of computer science from the one presented in your classes, but one which is just as rich, if not richer, in elegance, applicability, modularity, joy, and understandability than the stuff you're learning.

Functional programming is an entirely different way of thinking about programs, and has an exceptionally rich literature in academia.

What's more, the industry has recently been embracing functional programming, with things like React and FRP being a new way to do UIs (although technically it is anything but new, this kind of stuff was elaborated in the 70s, but nobody knows this).

However, functional programming isn't only used on the frontend. The separation between code and data offers modularity when doing data transformations. Continuation passing offers wonderfully expressive and powerful abstractions (which we might learn about at a later date). Advanced type theory (which is a branch of functional proramming, effectively) is used to prove properties of programs to ensure they are memory safe (no leaks and/or use-after-frees and/or null dereference and/or aliasing). The list of applications of functional programming is too long to list, because it is the the same applications as any other paradigm, i.e., programming in its entirety.

Thinking functionally lets you be exceptionally productive while ensuring that the programs you write will be modular and effective. You won't ever write spaghetti code, because spaghetti code is in essence a problem of shared state. Functional programming forces you to be explicit in all state that you use and doesn't let you change things willy-nilly for no reason.

1.1 Why Scheme?

Scheme is a language that is nearly as old as computers themselves. Scheme is a dialect of LISP, a language introduced by John McCarthy in the late 1950s.

It can be argued whether he "invented" it, or whether he "discovered" it. LISP is a language with incredibly simple syntax (much simpler than Java, Python, or any other language that you might use day to day) because it has a single kind of expression though which everything is built.

The syntax of lisp is composed entirely of two things: Literal objects (such as numbers or strings), symbols (meaning, words, such as variable names), and statements called "S-expressions" meaning "symbolic-expressions", but which are often abreviated to "s-exps"

Examples of objects are:

```
12345       ;; A number
12.52       ;; a floating point number
#\f          ;; the character "f"
#(1 2 3 4) ;; a vector constant
#t #f       ;; booleans
"string"    ;; string
'(a b c d) ;; a literal list of symbols
```

In scheme, a s-exp can either signify the calling of a function, or a literal list. If there's no special ' symbol before the list, then it's a function or macro (we'll get into these later) call.

A s-exp looks like the following:

```
(f a b)
```

This s-exp signifies the calling of the function "f" with arguments "a" and "b".

The only rule you need to know about s-exps is this very easy transformation:

```
func(a1, a2, a3)

gets turned into

(func a1 a2 a3)
```

And that's literally the only difference between reading scheme code and reading any other code.

Every function is called like this, including numeric functions, such as multiplication and addition. This can be a source of great joy when you no longer have errors due to order of operations (because the order of your operations are explicit).

Why do you want this? Spot the bug:

```
if (hamming_weight(c1 & 0b1010) + hamming_weight(c2 & 0b1010) % 2) \
   == (hamming_weight(p) % 2):
    key_biases[n1 | (n2 << 4)] += 1
```

1.2 Ok, but why?

In LISP, a list is written like this:

```
(a b c d)
```

And, calling the function `a` with arguments `b c d` is written like this:

```
(a b c d)
```

See the difference? There is none.

Why is this important? Because in LISP, code is literally data. You can write functions which take s-exps, and output code. Those are called macros.

In return for writing your code in regular syntax, you get free exceptionally powerful metaprogramming, and all you need to be able to do to take advantage of it is use list-processing functions.

1.3 Lambda

What is lambda? It is a notation for creating functions. In the 1930s by Alonzo Church as part of his research into the foundations of mathematics.

Lambda calculus is an exceptionally small language, but with it, we can write programs which are equivalent to a turing machine. In particular, infinite loops and stuff.

This language has only three components.

For one, there are variables, such as x, y, z.

For two, there are λ-abstractions, which are literal function definitions. Here's a function that takes one argument x and returns $x + 2$.

$\lambda x.(x + 2)$

And finally, there are "applications", meaning the application of a function to some argument. The argument could be whatever you'd like.

This is directly supported in scheme via the `lambda` function. It looks like this.

```
(lambda (a) (* a a))
```

This function takes an argument `a` and returns the square of `a`.

A lambda in scheme takes any number of arguments, but they're divided into a so called "lambda-list" meaning the list of arguments that the function takes, and then the rest of the function, which is its body.

```
(lambda (arg1 arg2)
  (function-evaluated-first arg1)
  (function-evaluated-second arg2)
  (function-evaluated-third arg1 arg2))
```

The last expression of a lambda is the one that is returned, so, in this case, it returns the result of the call `(function-evaluated-third arg1 arg2)`.

Once you have a function, you can pass it around, call it, wrap it in another function, put it in a list, a dictionary, etc.

1.4 So, what does scheme code actually look like?

In a scheme file, we use "define" in the same place as you'd use `def` in python.

Let's look at some example code. The classic example, the fibonnaci numbers.

```
(define fib
  (lambda (n)
    (cond
     ((= n 0) 0)
     ((= n 1) 1)
     (#t (+ (fib (- n 1))
            (fib (- n 2)))))))
```

This defines fibonacci as you might imagine. But, just like you might also imagine, has exponential runtime.

```
(fib 1)
$2 = 1
scheme@(guile-user)> (fib 2)
$3 = 1
scheme@(guile-user)> (fib 3)
$4 = 2
```

```
[snip]
scheme@(guile-user)> (fib 15)
$9 = 610
scheme@(guile-user)> (fib 30)
$10 = 832040
scheme@(guile-user)> (fib 40)
ERROR: In procedure scm-error:
User interrupt

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guile-user) [1]>
```

Let's interactively do this on my laptop.

1.5 Let's make this efficient, but functionally.

```
(define fib
  (lambda (n)
    (if (< n 1)
        (error "invalid input"))
    (letrec
        ((loop (lambda (i acc prev)
                 (if (= n i)
                     acc ;; if true
                     ;; else
                     (loop (+ i 1)
                           (+ acc prev)
                           acc)))))
      (loop 1 1 0))))
```

And this works as you'd expect.

```
scheme@(guile-user) [1]> (fib 1)
$11 = 1
scheme@(guile-user) [1]> (fib 0)
ERROR: In procedure scm-error:
invalid input

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guile-user) [2]> (fib 40)
$16 = 102334155
```

Why is this faster? It uses something called tail-call optimization.
Tail-call optimization is what lets us write loops in a functional style which are just as fast as iteration.

2 Tail-call optimization

The essential idea behind tail-call optimization is that if the last thing you do in a function is call another function, instead of calling that function, getting the value, and then returning it to your caller, LISP can trash all your state (because you're done your function) and then call the next function overtop of the stack space you were using.

In our case, since the arguments to `loop` are stored on the stack, when we call `loop` from inside loop, instead of pushing a new frame onto the stack, we overwrite the variables on the stack and then simply jump to where we were previously. In fact, let's disassemble the function interactively to see how it was compiled.

[aside about function-caling on the whiteboard]

3 Wrapping up

This is a lot of stuff for this week, we've introduced the entirety of scheme syntax, lambda, talked about lisp, and we're getting on our way to write programs functionally.