SI 413, Unit 3: Advanced Scheme

Daniel S. Roche (roche@usna.edu)

Fall 2018

1 Pure Functional Programming

Readings for this section: PLP, Sections 10.7 and 10.8

Remember there are two important characteristics of a "pure" functional programming language:

- *Referential Transparency.* This fancy term just means that, for any expression in our program, the result of evaluating it will always be the same.

  In fact, any referentially transparent expression could be replaced with its value (that is, the result of evaluating it) without changing the program whatsoever.

  Notice that imperative programming is about as far away from this as possible. For example, consider the C++ for loop:

  ```
  for (int i = 0; i < 10; ++i) {
    /* some stuff */
  }
  ```

  What can we say about the "stuff" in the body of the loop? Well, it had better *not* be referentially transparent. If it is, then there's no point in running over it 10 times!

- *Functions are First Class.* Another way of saying this is that functions are data, just like any number or list. Functions are values, in fact! The specific privileges that a function earns by virtue of being first class include:

  1) Functions can be given names.

     This is not a big deal; we can name functions in pretty much every programming language. In Scheme this just means we can do

     ```
     (define (f x) (* x 3))
     ```

  2) Functions can be arguments to other functions.

     This is what you started to get into at the end of Lab 2. For starters, there's the basic predicate procedure?:

     ```
     (procedure? +)   ; #t
     (procedure? 10)  ; #f
     (procedure? procedure?) ; #t
     ```

     And then there are "higher-order functions" like map and apply:

     ```
     (apply max (list 5 3 10 4)) ; 10
     (map sqrt (list 16 9 64))   ; '(4 3 8)
     ```

What makes the functions "higher-order" is that one of their arguments is itself another function. Functions that you write can also take procedures as arguments, just like any other argument.

This is neat, but again doesn't make Scheme *that* special. Something very similar is even possible in C by passing a function pointer to another function, albeit with a much more annoying syntax. And Java allows something kind of like this in its anonymous classes.

So really what makes Scheme special here is *syntax*: passing functions around is so easy, you can do it without even noticing!

3) Functions can be returned by functions.

   This means that, when functions are first-class, we can make "function factories", functions that produce other functions. Again, in many languages (even C) it is possible to return other *pre-existing* functions, but functional languages go one step further to allow the return of *nameless* functions. That is, functions can be created on the fly and returned. We'll see what this is about very shortly.

   (And yes, this is a very special thing! You won't see it in many non-functional programming languages.)

4) Functions can be stored in data structures.

   When functions are first-class, we can make lists or even trees of functions, just like a list of numbers or any other data structure. Again, remember that first-class status means that *functions are data.* In fact, later in this unit, we'll see how functions can actually represent entire objects like we would in Java or C++.

2 Lambda

Readings for this section: SICP Section 1.3 and PLP Sections 10.5 and 10.6

The *Lambda Calculus* is a way of expressing mathematics and computation, all with functions, developed by Alonzo Church in the 1930s. It was actually a competing idea with Alan Turing's universal (Turing) machine for the best way to express what is computable. Ultimately, it was proven that the two models are actually equivalent. The famous "Church-Turing Thesis", which is one of those things that everyone believes but cannot be proven, states that every possible model of computation is also equivalent in power to Turing machines.

Math geekery aside, the basic principles of Church's Lambda Calculus find their way into modern programming languages like Scheme: Functional recursion is important and useful, functions are data, and not every function needs to have a name.

To honor this mathematical tradition, many programming languages that allow the creation of nameless functions use the keyword (or function name) lambda to do it. For example, let's say we want to create a function that takes an argument x and adds 3 to it. Here's how you would write that in Python:

```
lambda x: x + 5
```

Or in Haskell (note that the slash is supposed to look like a Greek λ symbol):

```
\x -> x + 5
```

Or in C# (no syntactic reference to lambda here):

```
x => x + 5
```

The new version of C++ even has it (again, no lambda involved):

```
[] (int x) { return x+5; }
```

And, finally, in Scheme (or Lisp):

```
(lambda (x) (+ x 5))
```

Now if you type this last one into DrRacket, what you get back is just #<procedure>, which is the interpreter telling you that this is a procedure (and it has no name).

To actually call this function, we do it the same way we would any other function in Scheme: by putting it first after an open parenthesis, followed by its argument:

```
((lambda (x)
   (+ x 5))
 8)
```

That function call now produces the answer, 13. I broke it up for you to see the structure, but we could of course also write it on a single line.

Now with this simple example, it's hard to see the utility of lambda and "anonymous" (i.e., nameless) functions in general. After all, the last expression above could certainly be accomplished more easily just by writing (+ 8 5)!

Well, we will see many examples of the power of lambda, but one useful purpose is in combination with higher-order functions like map and apply. For example, we can use the function above to add 5 to every element in a list:

```
(map (lambda (x) (+ x 5))
     (list 3 5 8 10))
```

Before we would have needed a recursive function (or in another language, a for loop) to accomplish the same thing. Now we can accomplish the same thing, with a very nice separation of the *action* (adding 5) from the *data* (the actual list).

One last note here on lambda: it lies beneath some of the Scheme code you've already been writing! For example, when you define a function like

```
(define (foo x y)
  (* (+ x 1) (- y 1)))
```

this is really equivalent to defining foo as a *constant* lambda-expression:

```
(define foo
  (lambda (x y)
    (* (+ x 1) (- y 1))))
```

You might be even more startled to discover that a let expression is also just a lambda in disguise. Think about it: let is just saying "associate these names with these values, and then evaluate this expression". Well that's precisely what it means to call a function! For example, this let expression:

```
(let ((x 8)
      (y 9))
  (+ (sqr x) (sqr y)))
```

is exactly the same as:

```
((lambda (x y)
   (+ (sqr x) (sqr y)))
 8
 9)
```

Really! They are equivalent! The argument list in the lambda is just the names of all the let variables, and the actual arguments are the values that each let variable gets assigned to.

These translations are not just for our entertainment, they are actually how the Scheme interpreter works. So function definitions are just a shorthand for a constant definition of a lambda expression, and let statements are a (very convenient) shorthand for a lambda evaluation. There is a term for this sort of thing: *syntactic sugar*. We say that function defines and lets are "syntactic sugar" in Scheme because they are convenient (and sweet!), but don't really provide any new capabilities to the language. In other words, they are empty calories. Delicious, useful, wonderful empty calories.

3 Side Effects

Readings for this section: SICP, Section 3.1.

Remember that one of the key principles of "pure" functional programming is *referential transparency*. What this means specifically is that any function call (or any expression) in our program could be replaced with the value that results from that function call, without changing the program at all.

Informally, what referential transparency means is that there are no side effects to any computation. Every function returns some value, and that is the only thing that happens. Nothing else changes. In particular, if we made the same function call twice, we are just wasting our time because the result will be identical.

Some programming languages, such as Haskell, enforce referential transparency completely. You must do pure functional programming in such languages. As a result, they can perform a number of optimizations to the code that would not be possible if side effects were happening. But on the other hand, doing some seemingly simple tasks suddenly becomes complicated because of the restrictions in the language.

Scheme is quite so pure as that. While Scheme is primarily designed to be a functional programming language, *you can break the rules* if you really want to. Now I'm going to tell you how to do it. This doesn't mean that you *should* break referential transparency all the time in your Scheme programs, just that you *can* if you have a really good reason.

3.1 Printing

The simplest kind of side-effect is printing some text to the screen. Now it is important to distinguish this from the return value of a function, which DrRacket also displays on the screen. Actually, DrRacket makes this distinction by formatting the output you ask for explicitly in purple, as opposed to the return value of the function which is printed in black.

Here are the important functions for printing in Scheme:

- (display X): Displays X just like the interpreter would print it. Note that X could really be *anything*: a number, a symbol, a list, . . .
- (newline): Prints a line break
- (printf "string" args ...) : Works similarly to the printf from C that you know and love. The difference is that instead of having to do %d or %s or %u for all the different types of arguments, you only need to do ~a for any kind of argument.

  Try (printf "Homework ~a\nis due ~a\n" 4 'tomorrow) for an example.

3.2 Extra Control Constructs

Now that we can break referential transparency, some other basic aspects of Scheme also get screwed up. To illustrate the difficulty, say we want to write a function that prints every element in a list:

```
(define (print-list L)
  (if (null? L)
; WHAT TO DO FOR BASE CASE???
(display (car L))
(newline)
(print-list (cdr L))
; HOW TO GET ALL THIS TO HAPPEN IN THE ELSE???
))
```

There are two problems here. First, what should we do in the base case? We don't want to print anything, so what should be returned? Some versions of Scheme allow us to create a "nothing" value called void, but this version of Scheme is simpler: we can have a 2-part if expression that has no "else" clause. In such an expression, if the if condition is false, nothing is returned. In the function here, this just means negating the base case test to (not (null? L)), and then eliminating the "else" part of the if.

(Note: some of you have done such things with if expressions already - although you probably didn't mean to! Now we see why Scheme allows things like (if (< x 15) 10) with no "else" part.)

The second issue is that we want to do more than one thing when the condition is true. When we have referential transparency, there's never any cause for this, because we aren't really *doing* anything other than computing the answer. But to allow for sequential execution with side-effects, Scheme has the begin special form. This just evaluates each of its arguments in turn, and returns the value of the last expression.

Now we can actually write the print−list function:

```
(define (print-list L)
  (if (not (null? L))
      (begin (display (car L))
             (newline)
             (print-list (cdr L)))))
```

3.3 Mutation

And now for the ultimate taboo. The true break of referential transparency: changing the value of an existing variable. So far, if you make a new variable with define or let, it's essentially a *constant*; its value will always be the same as whatever you initially defined it to be. As we discussed, this supports referential transparency, allows certain compiler optimizations, and is generally the *preferred way* of writing Scheme code.

But there is another way. Although Scheme is a "functional-first" language, in which the very best way to code is using referential transparency, it it still possible to "break the rules" and change things. And the ultimate rule-breaker is set!:

```
(define x 10)     ; now x is 10, of course
(set! x 11)       ; Great Scot! x is changed to 11
(set! x (* x 2))  ; And now x is 22
```

To emphasize, *you don't normally want to do this*, and for this class in particular, you had better have a really good reason before you start throwing set!s around in Scheme. Here's a really good reason:

3.4 Closures

As you learned in Lab 3, a closure is simply a function plus its referencing environment. When we define a function within the context of some local variables, this means that that function maintains copies of those local variables as part of its *state*, the state of the referencing environment of the closure! In particular, in Scheme, when you put a lambda inside of a let, the variables from the let persist through different calls to that lambda function. So if you update those values with the set!, you have something very much like objects in Java or any other language.

Enough words, how about an example. Let's say we want a "counter" to count how many times it is called. We could do this with a global variable as follows:

```
(define count 0)
(define (count-inc)
  (set! count (+ count 1)))
```

This works, but it has a weakness in that it relies on a global variable count. If you want to count two different things in the same program, you're out of luck, since there is only a *single* global variable for the count.

But here's where closures show their true power. Instead of a global variable and a global increment function, we can make a *function factory* with its own local variable and local increment function, as follows:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      count)))
```

Now we can get a brand new counter increment function every time we call (make—counter), and this returned function will increase and then return the current count. The magic here is that because the let is inside the function factory definition, the count variable is *local* to that function call and to the lambda function that gets returned. In other words, each call to (make—counter) makes a brand new closure with its very own count variable - very much like calling "new" in Java to construct an object. So we can have multiple counters going at the same time, independently, no problem at all.

In Lab 3, you see some more sophisticated uses of this concept, but it's all essentially the same. I just want to take this opportunity once again to emphasize that set! is a powerful tool in Scheme, not to be used often, but in this very particular kind of case it is quite beneficial.

4 Efficiency

Readings for this section: PLP Section 6.6.1 and SICP Section 5.4.2.

In any programming language, it is very important to be able to accomplish tasks *efficiently.* In a low-level language like C, this is always going to be possible, although it might require some more programming work to make it happen. But in high-level languages, especially *declarative* languages where the translation to machine code is not as obvious, we still have to make sure that our code is fast.

4.1 Memoization

Consider the classic example of the Fibonacci function:

```
(define (fib n)
  (if (<= n 1)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

This implementation is correct, but really *really* slow. For example, try computing (fib 40). The problem is that there are two recursive calls, resulting in an exponential explosion of the running time.

But think about it - how many recursive calls do we really need to compute (fib 40)? All we really need is the 40 previous Fibonacci values. If we could just remember them when they get computed...

Well that's exactly what the technique called *memoization* gives us. The basic idea (recall from algorithms class) is that we will save the results of every function call in a hash table. Then whenever a new call is made, we first check to see if the result is already stored, and if so just return it and avoid any computation.

In order to do that in Scheme or any language, we need hash tables (or some other kind of fast-lookup dictionary). Yes, Scheme has hashtables! Here are the most important functions:

- (make−eqv−hashtable): Creates and returns a new, empty hash table.
- (hashtable−set! h k v): Modifies the hash table h so that key k is set to value v. (Notice how the ! convention is used for this function name, since the function has a side-effect of modifying the hash table h.)
- (hashtable−contains? h k): A predicate to check whether the hash table h has any value associated to key k.
- (hashtable−ref h k d): Given a hash table h and key k, returns the value associated with that key. If there is no such key in the hash table, the default value d is returned instead.

Now here's how we can memoize the Fibonacci function. Look carefully: fib−memo is not a function definition, but rather a *constant* definition with a let inside it, and inside that let is the actual memoized function definition. The reason to do it this way is to avoid creating a global variable for the table itself. In other words, after this definition, there is just one new name fib−memo in the global namespace, and because of scoping rules *only that function body* is able to touch the memo hash table.

```
(define fib-memo
  (let ((memo (make-eqv-hashtable)))
    (define (fib-internal n)
      (cond [(<= n 1) n]
            [(hashtable-contains? memo n)
             (hashtable-ref memo n '())]
            [else
             (let ((val (+ (fib-internal (- n 1))
                           (fib-internal (- n 2)))))
               (hashtable-set! memo n val)
               val)]))
    fib-internal))
```

4.2 Tail Recursion

Let's say that, for some reason, we want to sum up the squares of all the numbers from 1 up to n, for any n. Here's a decent way to write that function in Scheme:

```
(define (ssq n)
    (if (= n 0)
        0
        (+ (sqr n) (ssq (- n 1)))))
```

Now doing something like (ssq 10) or even (ssq 1000) works fine. But if we want to go really big, like (ssq 4000000), the interpreter runs out of memory.

Now 4 million is a big number, so maybe this is just impossible for a computer. But if you wrote this function in Python, like

```
def ssq(n):
    sum = 0
    i = 1
    while i <= n:
        sum = sum + i*i
        i = i + 1
    return sum
```

and then called ssq(4000000), it would return the result in less than a second and not use more than a few kilobytes of memory. So what's going on? Is Scheme inherently slower than other languages - even other *interpreted* languages like Python - on this kind of problem?

The problem with our Scheme function is that it is recursive, and there is a memory overhead for each recursive call. As each call is made, a new activation frame is pushed onto the stack with information such as the argument values. And this is absolutely necessary, since the computation has to successively pop all these values off the stack and then add (sqr n) to them before returning to the next level.

It would seem that the imperative programming way of storing the running sum in a local variable is the "right" way to do this, but how could we do this in Scheme, with referential transparency? The answer is to build this local *accumulator* variable into the recursive function as an extra argument. In each recursive call, this argument will keep track of the running sum, or whatever else we're computing. Here's how the previous function could work:

```
(define (ssq-helper n accum)
  (if (= n 0)
      accum
      (ssq-helper (- n 1)
                  (+ (sqr n) accum))))

(define (ssq n) (ssq-helper n 0))
```

See the difference? In the new helper function, *there is nothing to do after the recursive call returns.* This means that the stack space for this recursive call can be re-used, without having to wait for the whole computation to finish.

More formally, we say that the recursive call in the ssq−helper function is in *tail position.* Tail position means the last statement of a let, define, or begin, or any clause of an if or cond, or a few other things. Since the ssq−helper recursive call is in tail position within the if, and the if is in tail position in the define, the recursive call is in tail position in the entire function. And when *every* recursive call is in tail position, the function is said to be *tail recursive.*

So what? Well this means that the Scheme interpreter can *optimize* this function call so that a bunch of extra stack space is not actually used for the recursive calls. Essentially, the Scheme interpreter will turn our recursive-looking code (tail recursive, that is) into the iterative code that we might write in C++ or Java. This is called *tail call optimization*, and it's actually a requirement of any Scheme interpreter or compiler to do it.

Here are some general guidelines for writing tail-recursive functions:

1) Write a helper function with an extra *accumulator* variable to store all the extra information associated with the recursion.
2) The base case return value of the helper function will just be the value stored up in the accumulator argument. (If it's a list, you will often have to reverse it.)
3) Make sure the helper function is tail recursive! The computation that you used to do outside the recursive call usually goes inside it now, in the accumulator argument.
4) The original function now becomes very simple: it just calls the helper function. The initial value of the accumulator is what used to be returned in the base case (usually 0 or null).