# SI 413, Unit 2: Scheme Basics

Daniel S. Roche (roche@usna.edu)

Fall 2018

## 1 Hello, Scheme

This unit's notes will make the most sense if you've already worked through Lab 1.

Scheme is a programming language whose history (based on another, earlier language called Lisp) goes back all the way to the 1950s.

The most distinctive thing about Scheme (or Lisp) is its *syntax*: everything you write in a Scheme program looks like

```
(function_name argument1 argument2 ...)
```

And when we say everything, we mean *everything*, from usual function calls to variable assignment, all kinds of numerical arithmetic, input/output, and even if statements and function definitions themselves.

Consider this block of code you might write in the second week of a C programming class:

There is actually a lot going on there, with a variety of different operators like +, <, =, as well as special keywords such as if and else and other syntactic symbols ;, { and }. This program looks fine to you only because *you know what all those things mean*, and they follow along with some intuition you have from math and the English language. But it's might not be obvious what all the steps are that are happening, and in what exact order they happen. (For example, if x is not a number but some special class instance, all of these operators could be doing something very different from what you might expect.)

Now consider the same exact program in Scheme:

```
(define z
  (if (< (+ x 1)
         y)
      (+ 3
         (* y 7))
      0))
```

This looks funky because it doesn't fit well with how we are used to writing math and English. But the rules — though perhaps tedious — are very easy to follow, as long as we can match up all the nested parenthesis (which proper indentation is crucial for!).

We don't have to "know" (or look up) the fact that $*$ has higher precedence than + in Scheme, because the language forces us to specify the order of evaluation no matter what. And calling a built-in operator like + looks no different than calling a function we wrote ourselves — in fact, you can even write your own operators in Scheme, no problem:

```
(define (% a b)
  (- a
     (* b
        (floor (/ a b)))))
```

Then, for example, (% 57 10) would evaluate to 7. This ease of *language extensibility* by writing our own useful functions that are indistinguishable from the built-in ones is one of the most attractive things about Scheme programming.

## 2 Lists and List Processing

**Readings for this section**: PLP, Section 10.3 through 10.3.2, and SICP, Section 2.2.1 up until the part on "Mapping".

Lists are the most important data structure in Scheme, and they are built right into the language and interpreter. These are actually linked lists, and they are built with the Scheme "glue" cons that you saw in Lab 1, as well as '(), a special constant that represents the empty list.

(By the way, other versions of Scheme allow you to write null or empty instead of '() for an empty list. Since we are using R5RS, only '() is allowed. This being Scheme, you could always (define null '()) if you like!)

With just cons and '(), you can build any old list. For example, the linked list with 1, 2, and 3 is created by the expression

(cons 1 (cons 2 (cons 3 '())))

To break apart lists and access their insides, we use the built-in functions car and cdr. Both of these can *only* be called on something which is a cons-pair; for example calling (cdr 1) will produce an error.

car gives you the first part of the cons pair, which also means (car L) produces the first element in the list L. cdr, its counterpart, produces the second part of the cons pair, which means (cdr L) is everything *except* the first element in the list L. **Important**: Keep in mind that car always produces a list *element*, while cdr always produces *another list*. Repeat that to yourself before you go to bed so you never forget it!

> (Aside) The names car and cdr are historical. The very first implementation of Lisp (Scheme's predecessor) was on a machine where every register could be split into the "address" part and the "decrement" part. Lisp used these two parts for cons pairs: the "Contents of the Address Register" and "Contents of the Decrement Register", hence "car" and "cdr".

Often we want to call a series of car's and cdr's. For example, to get the third element out of a list L, we would write

(car (cdr (cdr L)))

This can be abbreviated by just using the sequences of a's and d's for the car's and cdr's, surrounded by a single 'c' and 'r'. For example, the expression above is the same as (caddr L). (However, these are only supported up to four or five letters in the middle. Then you have to write it yourself...)

You can see that after a while, building lists with cons gets pretty annoying. To alleviate this annoyance, two extremely useful list-building functions are list and append.

(list 1 2 3) produces with list with those three elements: (1 2 3). In general, list puts all its arguments into a single list.

(append (list 1 2 3) (list 3 4) (list 5 6 7)) produces the single list (1 2 3 4 5 6 7). In general, append *concatenates* its arguments into a single list.

Please take a moment to make sure you understand the difference between these two functions. The key difference is that the arguments to list can be anything; they just become the elements of the new list. The arguments to append, on the other hand, must already be lists.

What you may notice in all the preceding discussion is that *types* are still very important in Scheme, even though it appears that "there are no types" in Scheme. In reality, *everything* in Scheme has a type; the difference with languages such as C++ and Java is just that Scheme does not require you to *declare* the type of each variable. This is oftentimes convenient for writing simple code quickly, but it leaves more responsibility to the programmer to ensure (for example) that append is never called on something which is not a list.

# 3 Recursive List Processing

In Scheme, we will be writing many functions that work on lists. In general, these will be written *recursively*. This is not really a hard requirement in Scheme, but it's just the way things tend to work better: recursive functions are very natural and simple to write. Evidence of this is that, for example, there is no while loop in Scheme.

If you disagree that recursion is more natural, many functional programming experts would say this is just because you learned procedural languages (C, Java, etc.) first. I'm not sure whether I (Dr. Roche) agree, but anyway I encourage you to give recursion a chance - it can be a nice, simple escape from the frustrating aspects of writing loops: do I want to start at 1 or 0? should that be less than or $\leq$? how do I avoid printing an extra space at the end? and so on.

The way a recursive list function works is to first check if we're in the base case (usually, when the list is empty). If so, we return whatever the function should return in this simple base case, generally some fixed value.

Otherwise, in the recursive case, there will usually be a recursive call on the cdr of the original list. This is because the cdr of the list contains everything except the first element. The result of this recursive call is usually then transformed in some way (perhaps making use of the car of the list) to get the overall answer to the function.

The only thing we need is a way of testing whether we are in the base case; i.e., whether the list is empty. For this, the predicate null? can be used; it returns true if and only if its argument is an empty list. The opposite function, pair?, is also useful sometimes.

This general pattern, which we will see over and over again in labs and homeworks, is as follows:

```
(define (list-fun L)
  (if (null? L)
      0 ; base case for an empty list goes here
      ; Recursive case goes next.
      ; Get the recursive call and do something with it!
      (+ 1 (list-fun (cdr L)))))
```

Hopefully you can see that this function actually computes the length of its input list L. And all we needed to write the function are two facts:

1) The length of an empty list is 0 (the *base case* return value).
2) The length of a non-empty list is 1 more than the length of the list with the first element removed (the *recursive case* return value).

If this all seems very confusing at the moment, don't worry too much; you'll get plenty of practice. Soon you'll be wondering why you ever wanted to use a for loop.

# 4 Quoting

**Readings for this section**: SICP, Section 2.3.1

Right now you know that Scheme has numbers, booleans, cons-pairs and lists, and you probably won't be too surprised to learn that it has strings too. But Scheme also has a different datatype called a "symbol" that can be rather useful.

Symbols are written by prefixing a word (the name of the symbol) with a single quote, like `'this`. A good way to think of symbols is as variable names, that don't refer to any value other than themselves. They're kind of like immutable (unchangeable) strings, except that they are usually short and almost never contain spaces. They're also sometimes used like enum types in C.

Symbols can be compared with the function eqv?, so for example

(eqv? 'Hello! 'Hello!)

returns true, while (eqv? 'this 'that) is false. You can also check whether some unknown thing is a symbol at all by using the symbol? predicate.

> Tip: eqv? can be used to compare all kinds of things besides symbols, including numbers, booleans, and cons-pairs. Try it out!

Symbols are a special case of a *quoted expression* in Scheme. Quoting is a way of saying "don't evaluate this" to the Scheme interpreter. And the single-quote notation like 'this is just a short-hand for using the quote function like (quote this).

So you see, a symbol is just a quoted identifier! But we can quote much more than identifiers; *any* expression in Scheme can be quoted. In fact, you will often use quoting as a convenient way of creating lists without using cons or list. To quote a list, you just add a single quote in front of it (or give it to the quote function). So for example, the four expressions

'(1 2 3), (quote (1 2 3)), (list 1 2 3), and (cons 1 (cons 2 (cons 3 '())))

all produce exactly the same thing. Quoting is in fact more powerful than the list function because it can also produce nested lists like '(1 2 (3 4) 5 6 ((7))). Producing this same nested list using cons or list would be much more painful.

And now you see why '() is the way that the empty list is written in Scheme too!

# 5 Let

**Readings for this section**: PLP, Section 10.3.4, and SICP, Section 3.1.1

As you already saw in Lab 2, let is a way of getting local variable definitions in Scheme. The way it works is like this:

```
(let ((a 3)
      (b 7))
  (* a b))
```

(which produces 21). In fact, the second part of each variable specification can be *any* Scheme expression, so you can also do stuff like:

```
(let ((x (* 3 4)))
  (+ x 8))
```

(which produces 20). Like everything else, lets can also be nested, which can even give the illusion of a variable value changing (although it doesn't, they're just different variables with the same name). For example, the following C code:

```
int a = 4;
int b = 12;
a = b - a;
return a * 2;
```

Could be translated as the following in Scheme:

```
(let ((a 4))
  (let ((b 12))
    (let ((a (- b a)))
      (* a 2))))
```

Now this isn't necessarily the best way to do things in Scheme, but anyway it's possible. And actually, there are some situations when using a local variable is the only reasonable approach. Consider for instance the following Scheme program:

```
(define (lmax L)
  (cond [(null? (cdr L))
         (car L)] ; if list has one element, return it
        [(>= (car L) (lmax (cdr L)))
         (car L)] ; if first element is largest, return it
        [else (lmax (cdr L))])) ; else return recursive call
```

What is the running time of this program, in the worst case?

Of course we could write a recurrence to figure that out, since this is in fact a recursive function. The worst-case cost is given by

$$T(n) = \begin{cases} 1, & x \leq 1 \\ 1 + 2T(n), & \text{otherwise} \end{cases}$$

And if you remember from Algorithms class, this turns out to be $O(2^n)$: exponential-time in the size of the input list!

The problem is that (lmax (cdr L)) is called twice every time in the worst case. This is a big waste of time, and there are a few ways to fix it. Probably the most obvious way is to use a local variable, like this:

```
(define (lmax L)
  (if (null? (cdr L))
      (car L)
      (let ((rest-max (lmax (cdr L))))
        (if (>= (car L) rest-max)
            (car L)
            rest-max))))
```

This is really the same thing as the code above, except that the embedded let avoids the need to ever make 2 recursive calls. The running time becomes just $O(n)$, as we would like.

# 6 Syntactic Building Blocks

There are four basic data components in any programming language:

- **Atoms**: Sometimes called *literals*, these are code fragments that can't be split up, need no further evaluation, and whose meaning never changes. Examples are literal numbers, characters, and booleans.
- **Values**: Code fragments that can't be evaluated any further, but which may be built up from smaller pieces. Every atom is a value, but so are things like lists, strings, arrays, and even class objects.
- **Expressions**: Code fragments that can be evaluated to produce some value. Now we are getting much more general to include things like variable names and function calls. We know these things will eventually produce or refer to some value, but that value needs to be computed as part of the program.
- **Statements**: A complete, stand-alone instruction or command in a program. Sometimes (like in Scheme), any expression is also a statement. In other languages (like C++ and Java), an expression *that is followed by a semicolon* becomes a statement. Programs in procedural languages, and even many other kinds, are simply a sequence of statements. Things like function definitions or loop bodies are *compound statements*, meaning they are statements made up of smaller statements.

In this terminology, a computer program is simply evaluating a bunch of expressions and executing a bunch of statements (which usually include expressions and/or other statements to evaluate and execute). In class,

we will look at some sample code fragments and identify which parts are atoms, values, expressions, and statements.

Now we can see one of the big features of Scheme is its *syntactic simplicity*. In Scheme, it's always clear what is a value or expression. And statements? Well, they're pretty much just expressions too. In fact, *a Scheme program is just a series of definitions and expressions*. And since even definitions look exactly like any other expression, this allows us to write a formal but very simple grammar for the Scheme language.

# 7 Evaluation Model in Scheme

**Readings for this section**: PLP, Section 10.3.5, and SICP, Section 1.1 through 1.1.4

You have probably noticed that a Scheme *program* looks very similar to scheme *data*. For example, here is a very simple Scheme program:

```
(+ (* 3 4) 5)
```

You could get the Scheme interpreter to print out *exactly* the same thing by typing

```
(list '+ (list '* 3 4) 5)
```

Even better, you could just quote the thing: '(+ (* 3 4) 5).

Now here's the mind-blowing part: this isn't just a coincidence! The internal representation of a Scheme program by the interpreter is exactly the same as what you get when you quote that same expression. Every function call is actually stored in the interpreter as a Scheme list .

For this reason, Scheme is called a *homoiconic* language. This fancy word just means that a Scheme program is stored internally using the same data structures (namely, lists) that the program itself uses. So by learning Scheme, we are also learning (somewhat) how the Scheme interpreter actually works! It also means that Scheme has very powerful facilities for *self-extension* of the language. In other words, if there's something you wish Scheme did differently, you can probably just write a rule for that new behavior right into your Scheme program itself! (Note that this only applies to the *semantics*, or meaning of the language, not to the *syntax*.)

## 7.1 The REPL

Now you're ready to learn how the Scheme interpreter actually works. It uses what's called a "read-eval-print loop", or REPL for short. This is very common in interpreted languages, and it works exactly as stated:

1) Read an expression from the user or from the program
2) Evaluate that expression
3) Print out the resulting value
4) Go to step 1

Since Scheme is homoiconic, the expression that is read in in step (1) is stored as a list in Scheme (unless it's already an atom). The evaluation function is called eval and we can use it, like

```
(eval (list '+ (list '* 3 4) 5))
```

Now run the line above in Scheme and you'll get the result of the evaluation, 17. (There is a function to "print" as needed for step (3), but I'm not going to tell you about it yet.)

To prove my point, here is a complete program in Scheme that will give you a REPL of sorts; it does the read and eval parts, and returns all the results in one big list.

```
; No arguments, since the function does its own reading
(define (repl)
  (let ((expr (read))) ; why do we HAVE to use a let here?
    (if (eof-object? expr)
```

```
        '()
        (cons (eval expr env)
              (repl)))))
; start the REPL
(repl)
```

## 7.2 How evaluation works

The way eval works is pretty straightforward in most cases. To evaluate a list of expressions like (e1 e2 e3 ...) , we first evaluate every expression in the list (recursively!) to get a list of values like (v1 v2 v3 ...) . Then the first value v1 (which had better be a function) is called with the rest of the values as its arguments.

And of course, there's another function to apply a function to a bunch of arguments; it's called apply. So

```
(apply - (list 8 3))
```

returns 5, as you might expect.

The only time evaluation doesn't work exactly like this is when the function is what's called a *special form*. Special forms look like functions, but the difference is that they don't evaluate all of their arguments before being called. For example, if is a special form, because it doesn't bother to evaluate the if/else clause that it doesn't need to evaluate. In particular, we can evaluate an expression like

```
(if #f (5) 6)
```

and it works fine (returning 6), even though evaluating (5) would be an error. If if were a regular function, both of its arguments would be evaluated first, and so the expression above would always give an error.

> Note: If if were *not* a special form, it would be impossible to define recursive functions on lists like we have been doing. Do you see why this is?

Other special forms we have seen are define, cond, and quote. quote is actually the *ultimate* special form, since all it does is refuse to evaluate its argument!

In this sense, quote is the exact opposite of eval. And eval is the ultimate *function*; it just returns its (evaluated) argument! Hopefully you can start to get a picture of how much we could do with some careful use of eval, apply, and quote. You could take your whole program, modify it in some way, and return another program, for example. Try doing that in Java!