

08 – Functional Paradigm and Scheme

Programming languages generally fall within one of the various *paradigms*. Paradigms describe how problems are solved using a language. Some of the most common paradigms are:

Object-Oriented:

Programs are built by defining objects and their interactions. Objects hold state info. Languages generally also support inheritance and polymorphism.
Examples: Java, C++, C#, Smalltalk, Eiffel, Visual Basic.

Procedural:

Programs are built by hierarchically defining the tasks that need to be done.
Examples: C, Pascal, Fortran, Cobol, and most versions of BASIC.

Functional:

Programs are built purely out of mathematical functions without state information.
Examples: LISP, Scheme, ML, Haskell, Erlang

Logic:

Programs are built by defining logical rules and goals.
The runtime environment tries to achieve the goals by logical deduction.
Examples: Prolog, ASP, Datalog, Florid, Logtalk

There are other paradigms as well.

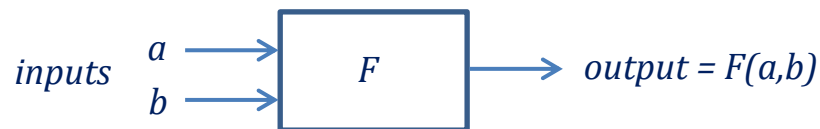
This section focuses on the **functional** paradigm. Some of the advantages and characteristics of functional programming are:

- Uniform view of programs as functions
- Ability to treat functions as data
- Limitations of side-effects
- Automatic memory management
- Easier to formally prove program correctness

Some drawbacks of functional programming are (1) efficiency, (2) difficulty in applying them to modern software design patterns. Although functional languages are not as widely used as are, say, object-oriented languages, their advantages have led many non-functional languages to include functional mechanisms to allow for functional programming.

We will study “pure” functional programming using the language Scheme. This will also enable us to learn a LISP-style language, as Scheme is very similar to LISP. Many languages utilize LISP-style list structures.

In a pure functional language, all tasks are specified as functions, which have inputs and outputs, but no side-effects. In math, a function's value depends *only* on its inputs. We sometimes call these “black box” functions, and this property is called *referential transparency*.



That implies that there is no persistent state information – that is, there are *no variables*. Instead, there are only values and identifiers for constants. We can assign a value into an identifier, but we cannot change it. Thus, functional languages are sometimes called *single assignment*.

The lack of variables has further implications. A language that does not support the ability to change state, also *cannot support loops*. This is because typical FOR, WHILE, and DO loops require a way of indicating when the loop should stop. Without any state changes, all loops would be infinite loops! So, pure functional languages also do not contain loops.

At first, the inability to use variables or loops probably seems extremely limiting – even disastrous! However, the paradigm does support *recursion*, and this, combined with conditional branching (IF and SWITCH) actually results in the same algorithmic power as procedural and object-oriented languages. It does mean, however, that we will be using recursion a LOT!

Elements of Scheme

In Scheme everything is expressed as a *list*. By “list”, we mean LISP-style parenthesized structures that can be nested. Here are some examples of lists (notice that the elements are delimited by spaces):

(1 3 8)	a list containing three integers
(1 3.7 "hi")	a list containing an integer, a real, and a string
(+ 3 8)	a list containing an operator and two integers
(18 (7 9) 83)	a list containing an integer, a list, and an integer
()	an empty list

When confronted with a list, Scheme tries to *execute* it. Scheme does this by assuming that the first element is a function, and the remaining elements are parameters. For example, for $(+ 3 8)$, Scheme returns 11.

Sometimes we want Scheme to execute our lists, and sometime we don't. When a list simply contains data, we don't want Scheme to execute it. For example, if Scheme tries to execute the list `(1 3 8)`, it will complain that "1" is not a function. If you don't want Scheme to execute your list, just place a single quote symbol in front of it, like this: `'(1 3 8)`.

Scheme has a large number of built-in functions. In this class, we will limit ourselves to these:

Math functions:	<code>+</code>	<code>-</code>	<code>/</code>	<code>*</code>	<code>modulo</code>	<code>expt</code>	<code>truncate</code>	<code>floor</code>	<code>ceiling</code>	<code>round</code>
Comparators:	<code>=</code>	<code><</code>	<code>></code>	<code><=</code>	<code>>=</code>	<code>equal?</code>				
Logical:	<code>if</code>	<code>cond</code>	<code>#t</code>	<code>#f</code>	<code>and</code>	<code>or</code>	<code>not</code>			
List processing:	<code>car</code>	<code>cdr</code>	<code>cons</code>	<code>null?</code>	<code>list?</code>	<code>list</code>				
Programming:	<code>define</code>	<code>lambda</code>	<code>let</code>	<code>display</code>	<code>newline</code>	<code>quote(')</code>				

Since Scheme expects the function being called to be at the front of a list, math-related tasks are expressed in "prefix". For example:

<code>(+ 3 8)</code>	returns 11
<code>(- (* 6 8) (+ 2 3))</code>	returns 43
<code>(modulo 9 2)</code>	returns 1 (in Java, this would be 9%2)
<code>(< 3 8)</code>	returns true (<code>#t</code>)
<code>(> 4 4)</code>	returns false (<code>#f</code>)

Building an "if" statement is done by using the "if" function, which expects three parameters. The first parameter is the condition to test. The second parameter specifies what to return if the test is TRUE. The third parameter specifies what to return if the test is FALSE. For example:

<code>(if (= 2 3) "hi" "bye")</code>	returns "bye" (because 2≠3)
<code>(if (= 3 3) (+ 8 9) (+ 3 4))</code>	returns 17

"cond" is another conditional function, similar to a switch statement. Consider the following example:

```
(cond ((= a 3) "hi")
      ((= a 5) "bye")
      (else "neither"))
```

If `a` was assigned the value 5, the above function would output "bye".

Writing Scheme Functions

To create a function in Scheme, we need to call the “define” function. “Define” expects two parameters: the first specifies the function’s signature, or prototype (i.e., it’s *name* and its *incoming parameter(s)*), and the second parameter specifies what the function should return. For example, here is a function that tests an incoming value and checks to see if it is odd or even:

```
(define (isEven x)
  (if (= (mod x 2) 0) #t #f))
```

Our function “isEven” has one parameter named x. It works by checking to see if x mod 2 is equal to 0. If so, then it returns TRUE (meaning that x is an even number). Otherwise it returns false. When Scheme “executes” our call to `define`, it builds our “isEven” function and we can now try to call it. For example, if we next ask Scheme to execute: `(isEven 43)`, it returns `#f`.

Note that our above function could have been written more compactly as:

```
(define (isEven x)
  (= (mod x 2) 0))
```

since the output of the “=” function is already the appropriate T/F value.

We can also write functions that compute numeric answers. For example, here is a function that computes the sum of squares of two inputs:

```
(define (sumOfSquares x y)
  (+ (* x x) (* y y)))
```

If we call this function with `(sumOfSquares 3 4)`, Scheme returns 25.

Some computations require recursion. Here is a recursive function definition that computes the factorial of an input parameter:

```
(define (factorial x)
  (if (= x 0) 1
      (* x (factorial (- x 1)))))
```

Note that we protected the base case with in “if”.

Also note the indenting we used for the if-then-else; placing the “else” case directly below the “then” case is a common practice among Scheme users.

List Processing in Scheme

We now turn our attention to the list-processing functions shown back on page 3. The three functions that form the basis of list processing are:

car cdr cons

The reason for these cryptic names is historical, but they are widely used and you will get used to them quickly. Here is what these functions do:

<code>car</code>	takes a list, and returns the first element of a list
<code>cdr</code>	takes a list, and returns an identical list with the first element removed
<code>cons</code>	takes an element and a list, and returns an identical list but with the item prepended onto the front.

Here are some examples:

<code>(car '(7 3 5))</code>	returns 7
<code>(cdr '(7 3 5))</code>	returns (3 5)
<code>(car '((7 8) (3 9) (2 6)))</code>	returns (7 8)
<code>(cdr '((7 8) (3 9) (2 6)))</code>	returns ((3 9) (2 6))
<code>(cons 6 '(8 9))</code>	returns (6 8 9)
<code>(cons '(6 7) '(8 9))</code>	returns ((6 7) 8 9)

Pay particular attention to the very last example. The `cons` function *is not* the same as concatenating two lists.

Note that the `cdr` and `cons` functions *always* return a list. The `car` function, however, returns an item of whatever type happens to be sitting in the first element of the list sent to it.

We can combine these functions in interesting ways. For example, if we want to extract the second element of list `L`, we can do it as follows:

```
(car (cdr L))
```

Very soon, we will need a way of testing whether a list is empty (often used for the base-case of a recursive function). The function `null?` does that:

```
(null? '(3 4)) returns #f, because the list '(3 4) is not empty.
```

Let's now dive into writing our own list processing functions.

Example 1 – Recall that the `car` function returns the first item in a list. Suppose what we really want is the *last* element in the list. Let's write a function called `lastElement` that, when called with something like this:

```
(lastElement '(3 4 5))
```

returns 5. Our strategy will be to repeatedly (recursively) call `cdr`, until the incoming list has only one item left. At that point, the `car` gives us our answer. How can we know when the list has exactly one item left in it? That is when the `cdr` of the list is empty.

```
(define (lastElement L)
  (if (null? (cdr L)) (car L)
      (lastElement (cdr L))))
```

It is instructive to trace the progress of this function on our example call, above. The initial call `(lastElement '(3 4 5))` will cause the recursive branch to be taken, because `(4 5)` is not empty. This causes the recursive call `(lastElement (4 5))` to be made, and then `(lastElement (5))`. At that point, `(cdr (5))` is indeed empty, and so the base case is taken. It then returns 5, because `(car (5))` is 5.

Example 2 – Let's write a function “`isInList`” that takes two parameters, an item and a list, and tests to see if the item is in the list. Our strategy will be to first check to see if it is at the front of the list, since we can easily use `car` to get the item at the front of the list. If not, then we can repeat this process for the `cdr` (recursively), until either we find the element, or the list becomes empty.

```
(define (isInList a L)
  (if (null? L) #f
      (if (= a (car L)) #t
          (isInList a (cdr L)))))
```

Note that this is an example of a *Boolean* function - it returns `#t` or `#f`.

The convention for writing if-then-else structures, with the “else” case directly below the “then” case, can become cumbersome if the statement is deeply nested. In the above example, we may opt for less indenting.

Example 3 – Let’s write an “append” function. Recall that `(cons a L)` prepends `a` onto the list `L`. We wish instead to write a function “appendTo” that takes an item `a`, a list `L`, and returns a list identical to `L`, but with the item `a` appended onto it. For example, we would like a call such as `(appendTo 3 '(8 9))` to return the list `(8 9 3)`.

It’s slightly trickier to write a function that returns a list. Since functional languages don’t allow us to modify variables, we *can’t actually change* the incoming list. We must build a *new list* out of the values sent into `a` and `L`. A good strategy for this sort of problem is to look at an example, and see how to build the answer. Here is an example call:

`(appendTo 3 '(7 8 9))`

In this example, `a` is 3, and `L` is the list `(7 8 9)`.

The desired answer is `(7 8 9 3)`. How can we build `(7 8 9 3)` out of `a` and `L`? The only function we have that can build a list is `cons`. Our answer could be built by a call such as `(cons 7 (8 9 3))`.

`(cons 7 '(8 9 3))`


`(car L)` can provide us with the 7. The list `(8 9 3)` is trickier... `(8 9)` is available to us as `(cdr L)`, and 3 is available to us as `a`. But how would we get the 3 onto the end of the `(8 9)`? Aha!... by using our `appendTo` function (the one we are writing!). That becomes our recursive case, like this: `(appendTo a (cdr L))`.

Notice that as we recurse, the list in the call keeps getting smaller. The base case would occur when the function finally receives an empty list. At that point we can simply `cons a` onto an empty list, to make a list out of it. The complete solution now becomes:

```
(define (appendTo a L)
  (if (null? L) (cons a '())
      (cons (car L) (appendTo a (cdr L)))))
```

Example 4 – Let’s write a “concatenate” function, that takes lists L and M as inputs and produces as output a single list that is the concatenation of L and M. Thus, if we call our function “concatn8”, a call such as:

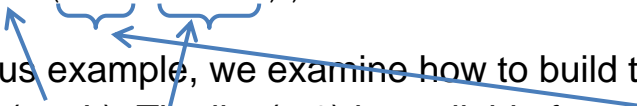
```
(concatn8 '(4 5 6) (1 2 3))
```



In this example, L is (4 5 6) and M is (1 2 3). In this case, our function should return the list (4 5 6 1 2 3).

We can use the same strategy as in Example 3. Looking at our example, we can build that solution using cons, as follows:

```
(cons 4 (5 6 1 2 3))
```



As we did in the previous example, we examine how to build these pieces. The 4 is available from (car L). The list (5 6) is available from (cdr L). And the (1 2 3) is, of course, simply M. How can we build (5 6 1 2 3) out of those two pieces? Again, by recursively calling our concatn8 function, like this: (cons (car L) (concatn8 (cdr L) M)).

This time, with each recursive call, it is the list L that keeps getting smaller. So our base case is when L is finally empty. When that happens, the concatenation of L and M is simply M. The completed solution is:

```
(define (concatn8 L M)
  (if (null? L) M
      (cons (car L) (concatn8 (cdr L) M))))
```

Higher-Order Functions

Most functional languages provide clean mechanisms for passing functions around as if they were data. There are two cases:

1. Functions as *input* to other functions
2. Functions as *output* from other functions

Let’s consider each case. Most students find the first case the easiest. These would be functions that have one or more input parameters that are, themselves, also functions. Let’s see some examples:

Functions as Inputs

Here is a function called “map” that takes two parameters, *f* (a function) and *L* (a list) as inputs. It then outputs a new list that is very similar to *L*, except that the function *f* has been applied to each item in the list:

```
(define (map f L)
  (if (null? L) '()
      (cons (f (car L)) (map f (cdr L)))))
```

Here is a diagram showing the effect of the map function:



To test our `map` function, we will need some input functions to send it. Here are some simple functions that we can create for testing purposes:

```
(define (cube x) (* x (* x x)))
(define (negate x) (- 0 x))
```

Now let's try calling `map` on a few cases, and watch what it returns:

```
(map cube '(2 5 3))    produces (8 125 27)
(map negate '(2 5 3))  produces (-2 -5 -3)
```

It is also interesting to see what happens when we pass a Boolean function into `map`. Recall the “isEven” function we wrote back on page 4. Applying it in the `map` function, we get:

```
(map isEven '(2 5 3))  produces (#t #f #f)
```

Functions as Outputs

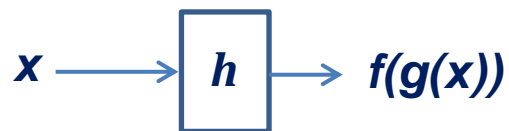
The second form of higher-order function is one that produces a function as its output. Such a function could be considered a sort of *function builder*. That is, based on the input(s) it receives, it builds and outputs a function that can be called or used at a later time.

Let's look at some simple examples of functions that return functions:

One example, commonly found in mathematics, is *function composition*. We can write a higher-order function in Scheme that performs function composition. Here is a diagram illustrating function composition:



Where the function $h(x)$ is equal to $f(g(x))$. That is:



Since `compose` is building a function as its output, one way of doing this is by using an *embedded* define. That is, we define our `compose` function in the usual manner, but inside the function it performs another define (to create the function labeled above as “h”, and then outputs that function:

```
(define (compose f g)
  (define (h x) (f (g x)))
  h
)
```

Note that `compose` doesn’t return $h(x)$, it returns *the function h itself*. So, how can we test our `compose` function? One way is to save the function that it produces – such as by calling it within a “define”:

```
(define H (compose negate cube))
```

Now we can actually call the generated function (we have saved it in the identifier `H`). A call such as `(H 10)` returns -1000.

Another way to test our `compose` function, is to place the call in a list – more specifically as the *first item* in a list. Then Scheme will attempt to call the function that `compose` produces. For example,

```
((compose negate cube) 10)
```

returns -1000.

Unnamed (“lambda”) Functions

In our `map` example, back on page 9, we created a couple of very small functions (“cube” and “negate”) simply to test our `map` function. It is possible to create these functions on the fly without assigning names to them. We commonly do this in our programs with numeric constants – for example, we often will write statements in Java such as:

```
for (i=1; i<10; i++)
```

wherein we have included two numeric constants (1 and 10) without giving them identifier names. We needed the values 1 and 10, so we just stated those values without storing them in variables.

Functional programming allows us to do the same thing with functions. This is another example of being able to treat functions as though they are values. The syntax for creating an unnamed function in Scheme is to replace the word “define” with the word “lambda”, and then leave off the name. Compare, for example, the way we defined our “cube” function, with an equivalent lambda function – both are shown here:

```
(define (cube x) (* x (* x x)))  
(lambda (x) (* x (* x x)))
```

The two functions are identical – they both have one input parameter `x`, and their outputs are specified exactly the same. The only difference, is that the first version is stored in the identifier “cube”.

We can pass lambda functions into higher-order functions. For example, in the “map” function we saw earlier, we could have passed in the cube function without bothering to store it first, as follows:

```
(map (lambda (x) (* x (* x x))) '(2 5 3))
```

Which still produces the same answer `(8 125 27)`, but without having to assign an identifier to the cube function.

Lambda also gives us another mechanism for writing functions that output functions. The “compose” function could have been written thusly:

```
(define (compose f g)  
  (lambda (x) (f (g x))))
```


Tail Recursion

One of the practical drawbacks of functional programming is performance. This is largely because of the increased need to use recursion. Each recursive call requires the runtime system to create an activation record (stack frame), which takes time and consumes memory.

In some cases, this problem can be mitigated by converting recursive calls to *tail recursion*. A function is tail recursive if its recursive call to itself is the very last operation that it performs. Let's see an example of converting a recursive function to one that is tail recursive.

Recall the function that we wrote earlier for computing factorial:

```
(define (factorial x)
  (if (= x 0) 1
      (* x (factorial (- x 1)))))
```

This solution isn't tail recursive, because the very last function performed is this multiplication.  Converting a function such as this to one that does the same thing but is tail recursive, can be tricky. The usual approach is to examine the operation(s) that happen *after* the recursive call, and if it is possible, to pre-compute it. The pre-computed value is then passed as an additional parameter (typically called an "accumulating parameter"). Since this changes the signature of the original function, we usually also have to create a helper function. A tail recursive version of the factorial function would then look like this (note the accumulating parameter "prod"):

```
(define (facthelp x prod)
  (if (= x 1) prod
      (facthelp (- x 1) (* x prod))))

(define (factorial i) (facthelp i 1))
```

The advantage of tail recursion is that an optimizing compiler can detect it and convert it to a loop, such as shown in the following pseudocode:

```
facthelp(x, prod)
do { if x==1 return prod
    x = x-1
    prod = x*prod }
```

Now at runtime the system need not generate stack frames for each iteration. Many compilers do this, such as the gnu C and C++ compilers.