Chapter 15

Functional Programming Languages

--- With an Introduction to Scheme

Contents

2. Higher-order Functions

The idea of higher-order functions is of central importance for the functional programming paradigm. As we shall see on this and the following pages, this stems from the fact that higher-order functions can be further generalized by accepting functions as parameters. In addition, higher-order functions may act as function generators, because they allow functions to be returned as the result from other functions.

Definition: A high-order function either takes functions as parameters or yields a function as its result, or both. These are, by definition, functions that operate on functions.

2.1 Classical (built-in) higher-order functions

apply

The function returns the result of applying its first argument to its second argument. For example,

```
(apply + '(7 5))          ;returns 12
(apply max '(3 7 2 9))     ; returns 9
```

map

The function map returns a list which is the result of applying its first argument to each element of its second argument.

```
(map odd? '(2 3 4 5 6))   ;Value: (#f #T #f #T #f)
```

The first argument can also be a user-defined function, for example

```
(map (lambda (x) (* x x)) '(1 2 3 4 5 6))
;return '(1 4 9 16 25 36)
```

filter

The function takes a list and generates a new list containing the members of the old list that match a user-specified condition. The implementation of the function is something like this:

We can then define a predicate and use it to filter:

```
(define (non-negative n)(>= n 0))

(filter non-negative '(-2 3 7 1)) ; return (3 7 1)
(filter non-negative '(100 -9 8)) ; return (100 8)
```

You can also generate the predicate function on the fly, for example,

```
(filter (lambda (x) (< x 0)) '(-2 3 7 1)) ; return (-2)
```

2.2 Functional composition

Function composition is a functional form that takes two functions as parameters and returns a function-- first applies the second parameter function to its parameter and then applies the first parameter function to the return value of the second parameter function. In other words, the function *h* is the composition function of *f* and *g* if *h(x) = f(g(x))*.

As a running example, we'll consider two functions:

f1(x) = x + 2

g1(x) = 3x + 4

These are modeled, respectively, by the following Scheme functions:

```
(define (f1  x) (+    x    2))
(define (g1 x) (+   (*   3   x  )   4))
```

Now functional h1 can be defined as follows:

```
(define (h1 x) (+ (+ (* 3 x) 4) 2))   ; f1(g1(x))
```

In **Scheme**, the functional composition can be naturally expressed as a higher-order function as follows:

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

For example, we could have the following:

```
((compose car cdr) '((a b) c d))
```

This call would yield c. This is an alternative, though less efficient, form of CADR.

Now consider another call to *compose*:

```
((compose cdr car) '((a b) c d))
```

This call would yield (b). This is an alternative to CDAR.

2.3. Using functions as arguments [2]

New scheme programmers often have trouble seeing places where passing functions as arguments would be useful. In many programs, there will be one or two instances of what is basically the same algorithm coded multiple times but with slight variations. These are perfect candidates for functions as arguments.

What you can do is code the basic structure of the algorithm into one function. Then, you can take the pieces that vary and add them back in as function parameters. This allows a great amount of customization within the function. You can also add extra function parameters for future expansion. Combining functions like this removes redundancy from your programs, and by extension, reduces the errors that come from re-coding the same algorithm over and over within a program.

Suppose you have **an order-processing algorithm** composed of several parts that:

- Processes each line of the order and adds up the total.
- Calculates shipping on the order.
- Validates the credit line of the purchaser.
- If successful, charges the order, sends an order confirmation, and records it in the database.

Now let's say that different customers have different types of shipping to calculate, have their credit line calculated differently, and are charged differently for each order. For example, shipping might be calculated through a different service provider depending on the client. The credit line might be checked through your own business on some customers or through the credit-card company on others. Order charging might vary depending on whether the client was normally billed, charged through their credit card, or performs automatic withdrawal. Different customers may have different processing needs for each of these stages.

One way to code such a function would be to simply hardcode all of the possible pathways in your order-processing algorithm directly. Then the call to this function would include a list of flags indicating which style of processing was requested. However, as the number of possibilities gets larger, the order-processing algorithm would become unwieldy.

Another way to code the function is to have these stages handled by independent functions passed to the algorithm. This way, the order-processing algorithm needs to have only the general flow of the algorithm coded directly. The specifics of each major stage would be handled by functions passed in. Such an algorithm would need to have parameters for the shipping-calculation function, the credit-validation function, and the order-charging function.

Parameters that are functions are passed just like any other parameter, the only difference being their use within the program. This technique allows you to have the algorithm code the general flow of control but still have the specifics of the processing parameterized. Many programmers who do not know this technique often have lots of flags that control the processing and that's not a bad idea when you have a few simple variations in processing. But it starts to get cumbersome as the number of differences grows.
It is difficult to determine at what point passing functions as parameters is more beneficial than having special cases and/or flags control the processing in a function. However, there are some guidelines:

- When the options are few and specific, special-casing is often the better method.
- When the options are so closely tied to the algorithm that it takes several functions using most of the algorithm's local variables to generate the desired options, special-casing is probably the better method.
- When the options are many or if new options are anticipated, functions as parameters is often the better method.
- When the options are very clear and logically separate from the code, functions as parameters is often the better method.

3. Examples of Scheme Functions

3.1. Smushing Together Two Lists

Scheme also has a standard function append that smushes together two lists.
For example:

```
(append    '(A B) ' (C D E))
;returns the list (A B C D E).
```

Similarly

```
(append    ' ((A B) C) '(D (E F)))
;returns ((A B) C D (E F)).
```

What is the functions implementation? Again, using pseudocode, the implementation looks something like this:

```
list append(lst1, lst2) {
        if (lst1 is an empty list) {
                return lst2
        }
        else {
                return cons(car(lst1), append(cdr(lst1),lst2))
        }
}
```

This pseudocode yields the Scheme implementation:

```
(define (append lst1 lst2)
      (cond
        ((null? lst1) lst2)
        (#t (cons (car lst1) (append (cdr lst1) lst2)))
))
```

As with length, the function append is already predefined in Scheme.

3.2 Finding the Maximum Element of a List

Let's next define a Scheme function max that accepts a nonempty list of integers as a parameter. The function then returns the maximum element of the list. Again, the pseudocode:

```
boolean max(lst){
        if (lst has only one element){
                return the one element in lst
        }
        else if (car(lst) > max(cdr(lst))) {
                return car(lst)
        }
        else {
                return max(cdr(lst))
        }
```

```
}
```

Problem: Write the Scheme function max.

3.3. Determining If Two Lists Are Equal

Next, let's define a Scheme function that determines whether or not two lists are equal (i.e. have the same elements).

What makes this problem hard is that a list may have a list as an element. So let's first focus on the easier case when the two lists consist of only atoms. We'll call such lists simple lists. In this case, the pseudocode looks like this:

```
boolean equalsimple(lst1, lst2){
  if (lst1 is empty){
     return true or false depending on whether lst2 is empty or not.
  }
 else if (lst2 is empty){
   return false
 }
 else if (first element of lst1 = first element of lst2){
   return equalsimple(CDR(lst1), CDR(lst2))
 }
 else {
   return false
 }
}
```

This pseudocode yields the following Scheme program:

```
(define (equalsimp lst1 lst2)
    (cond
        ((null? lst1)(null? lst2))
        ((null? lst2) #f)
        ((eqv?  (car lst1) (car lst2))(equalsimp (cdr
  lst1)(cdr lst2)))
        (#t #f)
))
```

Next, we look at more general lists, where the elements themselves can be sublists. This is an example of where recursion really shines. Any time the corresponding elements of the two lists are lists, they are separated into their two parts, car and cdr. We then use recursion on each part. Again, the pseudocode:

```
boolean equallist(lst1, lst2){
  if (lst1 is an atom){
     return true or false according to whether lst2 is a single atom that is equal to lst1.
  }
  else if (lst2 is an atom){
     return false
  }
  else if (lst1 is an empty list){
     return true or false according to whether lst2 is an empty list or not.
  }
  else if (lst2 is an empty list){
    return false
  }
  else if (equallist(CAR(lst1), CAR(lst2))){
     return equallist(CDR(lst1), CDR(lst2))
  }
  else {
     return false
  }
}
```

This pseudocode yields the Scheme code:

```
(DEFINE (equallist lst1 lst2)
    (cond
        ((not (list? lst1)) (eqv? lst1 lst2))
        ((not (list? lst2)) #f)
        ((null? lst1) (null? lst2))
        ((null? lst2) #f)
        ((equallist (car lst1)(car lst2))
          (equallist (cdr lst1)(cdr lst2)))
```

```
        (#t #f)
))
```

3.4. Sort a simple number list

A Scheme function that takes a simple list of numbers as its parameter and returns the list with the numbers in ascending order, e.g. (sort '(6 5 4 3 2 1)) returns (1 2 3 4 5 6). This implementation uses insertion sort.

```
(define (srt lst)
  (cond
    ((null? lst) lst)
    (#t (insrt (car lst) (srt (cdr lst))))
))
(define (insrt item lst)
  (cond
    ((null? lst) (list item))
    ((< item (car lst)) (cons item lst))
    (#t (cons (car lst) (insrt item (cdr lst))))
))
```

3.5. Finding the Number of Atoms in a List

Problem: Write a Scheme function numofatoms that determines the number of symbols in a list. For example,

(numOfSymbols '((1 3) 7 (4 (5 2)))) should return 6.

References:

1. Higher order functions,
http://www.ibm.com/developerworks/linux/library/l-highfunc.html

2. Lecture notes about functional programming in Scheme, http://www.cs.aau.dk/~normark/current-pp-notes-error-page.html

http://www.cs.caltech.edu/courses/cs1/resources/scheme-for-c-programmers.html
http://icem.folkwang-hochschule.de/~finnendahl/cm_kurse/doc/schintro/schintro_toc.html
http://home.adelphi.edu/sbloch/class/archive/160/fall2008/language/grammar.html