Introduction to List Processing and Functional Programming and Scheme

Theme

Introduction to functional programming using scheme (a dialect of lisp). In functional programming, programs are treated as function (for every input there is a unique output. In pure functional programming there are no variables and hence no assignment or side effects. There are function definitions and function applications and the value of a function only depends on the values of its arguments (referential transparency). Furthermore, the value of a function can not depend on the order in which its arguments are evaluated. A consequence of not having variables is that there are no loops; repeated operations and other control flow is obtained through recursion. Finally, functions are treated as first class values, i.e. functions are viewed as values which can be computed by other functions and passed as parameters to functions.

The lack of assignment and referential transparancy make the semantics of functional programs straightforward. There is no state, there is no concept of memory locations and pointers, only bindings of names to values - once a name enters the environment its value can never change. Such semantics is called *value semantics*. While this seems like a very restricted type of programming, it can be shown to be turing complete and hence, theoretically, as powerful as programs with loops and variables.

In practice functional languages, like lisp, scheme, and ML, are not pure; however, they are conducive to techniques that arise in pure functional programming and consequently provide the programmer with a different, and at times very useful, view of programming.

Topics

1. S-Expressions and Evaluation Rules

   Programs and data are the same - in scheme they are lists - the only difference is how they are interpreted. Given a list, S-Expression, (E1 E2 ... En), its value is obtained by recursively evaluating the S-Expressions E1, E2,...,En and applying the value of E1, which should be a function, with its arguments bound to the values of E2,...,En. In the base case, atoms, evaluate to themselves. This evaluation process is called applicative normal order. For example

   - (+ 2 (* 3 4)) => (+ 2 12) => 14 since + and * evaluate to the addition and multiplication functions and the numbers evaluate to their numeric values.
   - (max 1 2 3) => 3, since max is bound to the max function.
   - (1 2 3) => error since 1 does not evaluate to a function.
   - (list 1 2 3) => (1 2 3), since list is bound to a function that forms a list from its arguments.

- (list 1 (2 3) 4) => error since recursive (2 3) evaluates to an error due to the previous problem.
- (list 1 (list 2 3) 4) => (1 (2 3) 4)

2. List processing functions

   Syntactically lists are enclosed in parentheses with elements separated by spaces - (a1 ... am). Elements of a list may be lists themselves. The null list is denoted by (). The null list is detected by the predicate null? - a predicate is a function that returns true or false (in scheme denoted by #t and #f respectively). Note that the null list is considered both a list and an atom and evaluates to itself. Given a set of atoms, all lists, of arbitrary order (i.e. nesting) can be built from () and the function (cons x y), which constructs a list whose first element is x and remaining elements are the elements of y. Technically y need not be a list (to be discussed below). The first element of a list is obtained with the function car and the remaining elements with the function cdr.

   - Let L = (1 2 3), this is accomplished with (define L (list 1 2 3)). the define expression binds the value of the expression given in the second argument to the name given in the first argument.
   - (cons 3 ()) => (3)
   - (cons 1 (cons 2 (cons 3 ()))) => (1 2 3)
   - (car L) => 1
   - (cdr L) => (2 3)
   - (null? L) => #f
   - (null? ()) => #t

   Many other list processing functions are available (see chapter 7 of the scheme reference manual that is part of the MIT-Scheme distribution): E.G. append, length and reverse. These functions, and many others, can be built from the primitive list processing functions car, cdr, null? and cons using recursion.

3. Lambda expressions

   A function in scheme is denoted by (lambda name (formal-parameters) body), and is called a lambda expression. Lambda expressions evaluate to a function which can then be applied.

   - **(define sqr (lambda (x) (* x x)))** binds the name **sqr** to the function that computes the square of the parameter **x**.
   - Since it is common to define a name to a lambda expression, the following short hand exists. **(define (sqr x) (* x x))**.

4. Conditional expressions

    - The if expression provides a mechanism to return different values based on the value of a boolean expression. The expression **(if cond val1 val2)** returns **val1** if **cond** evaluates to **#t** and **val2** if **cond** evaluates to **#f**
    - (if (< 2 3) 0 1) returns 0
    - (if (< 3 2) 0 1) returns 1
    - The **cond** expression selectively returns a value based on a sequence of boolean expressions **(cond ((cond_1) val_1) ... ((cond_n) val_n))** is evaluated sequentially. First **cond_1** is evaluated and if true, **val_1** is returned. If false, then **cond_2** is evaluated and if true **cond_2** is returned. This process is continued until an expression evaluates to true. If none of the expressions is true, then what is returned is undefined. Note that the keyword else can be used for a condition and it always evaluates to true.

5. Recursion

    Since there are no side effects in functional programming, functions can be thought of as mathematical definitions and the use of recursive definitions and recursion allows us to define and hence implement many useful functions. Moreover, without side effects, there can be no loops since loops require a loop index which is incremented. Hence, in a pure functional language, all control must be done by recursion. Note that scheme is not a pure functional language, i.e. there are variables, however, we will focus on the functional programming style and use recursion for control.

    - The following recursive function computes n factorial. **(define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))**
    - The following recursive function returns a list of n ones. **(define ones (lambda (n) (if (= n 0) () (cons 1 (ones (- n 1))))))**

6. Higher-order functions

    Functions that take functions as parameters and functions that return functions.

    - sort: sort a list of objects using a specified comparison function
        - (sort '(4 3 2 1) <) => (1 2 3 4)
        - (sort '("one" "two" "three" "four") string< ?) => ("four" "one" "three" "two")
    - map: map a function over the elements of a list or lists. Given a function f of t arguments, (map f list_1 ... list_t), where list_1 = (a_11 ... a_1n), ..., list_t = (a_t1 ... a_tn), produces the list ((f a_11 ... a_1n) ... (f a_1n ... a_tn)). Examples:
        - (define sqr (lambda (n) (* n n)))
        - (map sqr '(1 2 3 4)) => (1 4 9 16)
        - (map (lambda (n) (* n n)) '(1 2 3 4)) => (1 4 9 16)
        - (map list '(1 2 3 4) '(1 4 9 16)) => ((1 1) (2 4) (3 9) (4 16))

- filtering lists - remove/retain the elements of a list satisfying a given predicate.
  - (keep-matching-items '(1 2 3 4 5) odd?) => (1 3 5)
  - (delete-matching-items '(1 2 3 4 5) odd?) => (2 4)
- reduce: combines the elements of a list using a given binary procedure. The combination procedes in a left associative order. (reduce f initial '(1 2 3 4)) => (f 4 (reduce f initial '(1 2 3))) => (f 4 (f 3 (f 2 1))). When the input is empty, initial is returned. reduce-right is the same as reduce, except a right associative order is used.
  - (reduce + 0 '(1 2 3 4)) => 10
  - (reduce * 1 '(1 2 3 4)) => 24
  - (reduce list () '(1 2 3 4)) => (4 (3 (2 1)))
  - (reduce-right list () '(1 2 3 4)) => (1 (2 (3 4)))

  The fold-left and fold-right commands are like reduce and reduce-right, except initial is always used. (fold-left f initial '(1 2 3 4)) => (f (fold-left f initial '(1 2 3)) 4) => (f (f (f (f ininitial 1) 2) 3) 4). fold-right is the right associative version.

  - (fold-left list () '(1 2 3 4)) => (((((() 1) 2) 3) 4)
  - (define (length list) (fold-left (lambda (sum element) (+ sum 1)) 0 list))
  - (define (reverse items) (fold-left (lambda (x y) (cons y x)) () items))
- function composition
  - (define (compose g f) (lambda (x) (g (f x))))
  - (define cadr (compose car cdr))
  - (define caddr (compose car (compose cdr cdr)))
  - (define cadar (compose car (compose cdr car)))
  - (cadr '(1 2 3)) => 2
  - (caddr '(1 2 3)) => 3
  - (cadar '((1 2) 3)) => 2
- currying - a function of multiple parameters can be viewed as a function of a single parameter that returns a function of the remaining parameters. For example, assume that f is a function of two variables.
  - (define (curry f b) (lambda (a) (f a b)))
  - (define plus1 (curry + 1))
  - (plus1 2) => 3