Functional languages

A functional language approaches programming from a more mathematical point of view than an imperative or OO language. Functional languages try to avoid the concept of state and instead focus on variables as representative of values rather than memory addresses.

The analysis of program semantics has already shown us that we can represent the meaning of a program using a functional representation. Functional languages enable a programmer to write directly in functional form. While it is not always obvious how to write a particular algorithm in functional form–just as it is not always obvious how to write some tasks in imperative form–functional and imperative languages are equally powerful in their capabilities.

The basis for functional languages is lambda calculus. A lambda expression, or lambda function is a nameless mathematical expression, where the mathematical expression becomes an object itself. As with typical substitution in mathematical expressions, variables in the expression can substitute for other variable names or for actual values.

Functional programming languages give the programmer a set of built-in functions, support various data types, and enable the definition of new functions. Loops are usually implemented recursively–as in our functional representation of loop semantics–and much of the language is built around lists.

Unlike pure mathematics, functional programming languages do permit state information and generally include an assignment statement. However, most functional language programmers try to avoid assignments, if at all possible.

Lambda expressions are not limited to functional programming languages. Python, for example, permits lambda expressions. In the example below, the lambda expression is assigned to a parameter of the function and then applied to the elements of a list.

```
# An example of using lambda functions in python

# imperative version that applies f to the elements of a
def applyFunction( f, a ):
    result = []
    for item in a:
        result.append( f(item) )

    return result

org = [1, 2, 3, 4, 5]

res1 = applyFunction( lambda x: 2*x, org )
print res1

res2 = applyFunction( lambda x: x*x, org )
print res2

expr = raw_input("Enter an expression of x: ");
res3 = applyFunction( eval('lambda x:' + expr), org )
print res3
```

The third case uses the eval function to permit the user to enter an arbitrary expression of x to apply to the list. Lambda functions permit the programmer to dynamically create functions of one or more variables and pass them around as objects.

Note that functional programming is not limited to functional languages. It is quite possible to write functional code in imperative languages that permit function return values. The example

below is an identical implementation of the applyFunction imperative version above. The functional version does not use a loop, but instead uses recursion to apply the function to the elements of the list.

```
# functional version
def applyFunctionRec(f, a, result = []):
    if a == []:
        return result
    else:
        return applyFunctionRec( f, a[1:], result + [f(a[0])] );

org = [1, 2, 3, 4, 5]

res1 = applyFunctionRec( lambda x: 2*x, org )
print res1

res2 = applyFunctionRec( lambda x: x*x, org )
print res2
```

Lisp/Scheme. Lisp and Scheme are some of the original functional languages, with Scheme being a derivative of Lisp. Expressions/statements in Scheme follow Cambridge-prefix notation with the function name given first, followed by the arguments, and the entire expression in parentheses.

```
(+ 8 5) : evaluates as 13
(* 4 6) : evaluates as 24
(- 8 3) : evaluates as 5
(+ 1 2 3 4 5) : evaluates as 15
```

The argument to any function can also be a function. We can define variables in Scheme using the define function, which puts a symbol into the global symbol table. Note, define is also how we can define new functions.

```
(define myvar 100)
```

Besides basic data types, the workhorse data type of functional languages is the list. Scheme has built in support for list creation and processing. The following shows some examples of list creation and manipulation. The traditional list manipulators are the `car` and `cdr` operators. The `car` of a list is the first element of the list. The `cdr` of a list is all of the list but the first element. The `cdr` of a list is always a list, or the empty list `()` if there are no remaining elements.

Example Scheme

```
: use a quote to specify that the list is not to be evaluated, but is a constant
(define fib '(1 1 2 3 5 8 13) )

(car fib) : evaluates to 1
(cdr fib) : evaluates to (1 2 3 5 8 13)
(car (cdr fib) ): evaluates to 1
(car (cdr (cdr fib ) ) ) : evaluates to 2
(caddr fib) : same as previous, evaluates to 2
(cdr (cdr fib) ): evaluates to (2 3 5 8 13)
(cddr fib): same as previous, evaluates to (2 3 5 8 13)
```

The `cons` and `append` functions enable building up lists from pieces. Append expects two list as arguments and returns their concatenation. Cons expects a value and a list and puts the value as the head of the list. Putting a non-list as the second argument to `cons` creates a non-list data structure called a dotted pair, which is generally not used in Lisp programming.

```
(append '(3 4) '(5 6) ) : evaluates to (3 4 5 6)
(cons 4 '(5 6) ) : evaluates to (4 5 6)
(cons '(3 4) '(5 6) ) : evaluates to ( (3 4) 5 6 )
```

Conditional control flow in Scheme uses if-then-else or a case statement. The function below, for example, defines a new symbol `backward` that reverses the order of a list.

```
(define (backward a)
  (if (null? a) () (append (backward (cdr a)) (list (car a)))))
```

The case syntax is similar to other languages. The first argument to case is the condition to test, the remaining arguments are the cases. In the example below, if the result of the test expression is 1, 2, or 3 then the first case executes. If the result is 10, 11, or 12, the second case executes. Otherwise, the else case executes. An else case is not required.

```
(display (case (+ 7 5)
            [(1 2 3) "weird\n"]
            [(10 11 12) "odd\n"]
            [else "whatever\n"]))
odd
```

There are several forms of iteration available in scheme, including for-loops, do-loops, and maps over lists. Python's for loop construct is identical to that of Scheme. The loop variable iterates of the list provided. As shown in the example below, the default `for-each` expression returns void as its result.

```
(for-each (lambda (x)
            (display (* x 2))
            (newline))
            '(1 2 3 4 5 6))
;; displays
;; 2
;; 4
;; 6
;; 8
;; 10
;; 12

(display (for-each (lambda (x) (* x 2)) '(1 2 3)))
(newline)
;; displays #<undef>
```

```
(display (map (lambda (x) (* x 2)) '(1 2 3 4 5 6)))
(newline)
;; displays (2 4 6 8 10 12)
```

There are many other forms of the for loop that return different kinds of values. For example, the `map` function shown above returns a list that is the concatenation of the values produced by the last expression in the body during each iteration.

The `do` function is more like a for loop in C. In the example below, the variable i is the loop variable. It receives the initial value 0 and the step-expression describes how the loop variable changes each iteration. The second argument is the termination condition; the loop will run so long as the termination condition is false. The remaining arguments form the body of the loop.

```
(do ([i 0 (+ i 1)])
     ((> i 5))
     (display i))
;; displays 012345
```

A map applies an expression to each element of a list and returns the new list. Lambda expressions are used within map expressions to define the operation applied to the list elements. They work identically to Python.

```
>  (define fib '(1 1 2 3 5 8 13))
>  (display (map (lambda (x)
                  (* x x))
                fib))
(1 1 4 9 25 64 169)
```

Scheme also includes functions for creating randomly-accessible data structures, such as arrays. In Scheme, the symbol `vector` creates a vector of items. The `vector-ref` function enables direct indexing into the structure. Therefore, we can write a simple recursive binary search on vector types, as shown below.

```
(define fibv (vector 1 1 2 3 5 8 13))
(define (binsearch v key lower upper)
  (let ( (mid (quotient (+ lower upper) 2) ) )
    (if (> lower upper) '()
        (if (= key (vector-ref v mid) key) (list (vector-ref v mid))
            (if (> key (vector-ref v mid)) (binsearch v key (+ mid 1) upper)
                (binsearch v key lower (- mid 1)))))))
(binsearch fibv 5 0 (- (vector-length fibv) 1))
(binsearch fibv 4 0 (- (vector-length fibv) 1))
```

Given the ability to assign values, execute conditional expressions, and loop, Scheme is clearly computationally equivalent to imperative languages. What is important to realize, however, is that functional languages are not intended to be written as imperative languages. They are optimized for tail recursion and functional solutions to problems. Many Scheme programs are a single expression, which often contain recursion.

The semantic meaning of Scheme programs is almost trivial to explain as a functional expression, since the Scheme program itself is a functional representation, even to the point of having functions like let and define that specifically add variables to the symbol table and define their scope.

Like other languages we have studied, Scheme has many built-in data types, mechanisms for catching exceptions, creating multiple threads, and reading and writing to the terminal, files, and ports. Scheme handles garbage collection automatically, although there are a set of functions the programmer can use to let the garbage collector know when a variable is no longer needed and to manually force garbage collection.

Scheme Syntax

General notes about syntax. Recall that Scheme is a dialect of Lisp, which stands for LISt Processing language. Scheme is designed to make it straight-forward to process lists.

- In general, every expression is surrounded by parentheses
- Comments begin with a semicolon
- Use prefix notation, e.g. (+ 3 5) evaluates to 8.
- booleans: #t and #f for true and false
- suppressing evaluation: a single quote can be placed before an expression to keep it from being evaluated. This is useful for making list literals.
- global variables use the define operator, e.g.

```
(define colors '(red yellow green))
```

Lists.

- To make a list literal, write a single quote, then put the list in parentheses
- To construct a list, use the **cons** operator. It takes 2 arguments, the second of which should be a list.

```
(cons 8 ()) ; evaluates to (8)
(cons 4 (cons 8 () )) ; evaluates to (4 8)
```

- To access part of a list, use operators that start with c and end with r. You can access the head of the list (an item) or the tail (the whole list without the head). But you can also apply the operation multiple times, e.g.

```
(define evens '(0 2 4 6 8))
(car evens)  ; evaluates to 0. this is the head.
(cdr evens) ; evaluates to (2 4 6 8). this is the tail.
(cddr evens) ; evaluates to (4 6 8). this is the tail of the tail
(cadr evens) ; evaluates to 2. this is the head of the tail.
```

- To concatenate N items together in a list, use the **list** operator, e.g.

```
(list 1 2 3 4) ; evaluates to (1 2 3 4)
(list '(1 2 3) '(4 5 6) '(7 8 9)); evaluates to ((1 2 3) (4 5 6) (7 8 9))
```

- To check to see if a list of null, use the **null**? operator, e.g.

```
(null? '()) ; evaluates to #t
(null? '(1)) ; evaluates to #f
(null? 5) ; evaluates to #f
```

- To check to see if two objects have the same structure and content, use **equal**? operator:

```
(equal? 5 5) ; evaluates to #t
(equal? '(1 2 3) '(1 2 3)) ; evaluates to #t
(equal? '(1 2 3) '(1 (2 3))) ; evaluates to #f
(equal? '() '()) ; evaluates to #t
```

Control flow.

- The if function has the form: (if <test> <then−part>) or (if <test> <then−part> <else−part>), e.g.

```
(define x 3)
(if (< x 0) -1 1) ; evaluates to 1
```

- The **case** function is like the switch in Java. The catch-all else case is optional. Here is an example:

```
(define month 'sep)
(case month
        ((sep apr jun nov) 30)
        ((feb) 28)
        (else 31))
```

Functions. Functions are defined using the define operator. You can define functions using a syntax that relates the language directly to the lambda calculus (i.e. by using the term lambda);

```
(define <name> (lambda (<arguments>) <function−body>))
```

e.g.

```
(define min (lambda (x y ) (if (< x y) x y)))
```

Or you can use a more compact notation

```
(define (<name> <arguments>) <function−body>)
```

e.g.

```
(define (min x y) (if (< x y) x y))
```