Chapter 11 - Functional Programming, Part I: Concepts and Scheme

Programming Languages: Principles and Practice, 2nd Ed.

Kenneth C. Louden

Introduction

- **Functional programming (F.P.) is in some ways the opposite of object-oriented programming: focus is on functions, all data is passive -- in OO the data name is first: x.f(), while in functional programming the function name is first: f(x).**
- **Functions can be used in F.P. like objects in OO: local functions are allowed, functions can be passed to other functions and returned as values, and functions can be created dynamically (functions are first-class values).**

Pure Functional Programming

- **No variables, only constant values, arguments, and returned values.**
- **No assignment or loops.**
- **All functions are referentially transparent: the value of a function depends only on the value of its parameters (arguments) and not on the order of evaluation or the execution path that led to the call.**
- **I/O is a problem, since any input function is *not* referentially transparent.**

Notes and Examples

- **Any referentially transparent function that has no parameters must always return the same value. So random() and getchar() are *not* referentially transparent.**
- **A purely functional sorting routine cannot sort an array in place -- it must create a new constant array with the sorted values.**
- **In order to program effectively in a purely functional way, constant creation must be as general as possible. In particular, *structured constants* must be available.**

Functional programming in an imperative language

- **Can one program functionally in C? Almost! The main problems are: insufficiently general functions, insufficiently general constants, problematic I/O.**
- **Why *would* one want to do so? Easier to understand, more reliable code (why?). But some situations are not suitable, e.g. sorting.**
- **What about Java? Is it possible to program functionally in Java?**
- **Yes! All objects must be immutable, and all variables and parameters must be final.**
- **Not really OO, since object state never changes.**

Sample functional style in C

- **The sum of integers up to a given integer, imperative:**

```
int sumto(int n)
{ int i, sum = 0;
  for(i = 1; i <= n; i++) sum += i;
  return sum;
}
```

- **Same function in functional style:**

```
int sumto(int n)
{ if (n <= 0) return 0;
  else return sumto(n-1) + n;
}
```

Functional style in C, continued:

- **Using tail recursion (recursive call occurs as the last operation) and a helper function:**

```
int sumto1(int n, int sum)
{ if (n <= 0) return sum;
  else return sumto1(n-1,sum+n);
}
int sumto(int n)
{ return sumto1(n,0);}
```

Accumulating parameter

Tail call

- **An optimizer can easily turn tail recursion back into a loop (so tail recursion can be more efficient):**

```
int sumto1(int n, int sum)
{ while (true)
    if (n <= 0) return sum;
    else { sum += n; n-=1; }  }
```

Functional style, continued:

- **The code of last slide changes addition order. With one more parameter we can preserve this order (useful for arrays and non-commutative ops):**

```
int sumto1(int n, int i, int sum)
{ if (i > n) return sum;
  else return sumto1(n,i+1,sum+i); }
int sumto(int n)
{ return sumto1(n,1,0); }
```

- **Array example (summing an array of ints in Java):**

```
static int sumto1(int[] a,int i,int sum)
{ if (i >= a.length) return sum;
  else return sumto1(a,i+1,sum+a[i]); }
static int sumto(int[] a)
{ return sumto1(a,0,0);}
```

Scheme: A Lisp dialect

- **Syntax (slightly simplified):**

  expression* → *atom* | *list

  atom* → *number* | *string* | *identifier* | *character* | *boolean

  ***list* → '(' *expression-sequence* ')'**

  expression-sequence
  → *expression expression-sequence* | *expression*

- **That's it! In Scheme, everything is an expression: programs, data, you name it. And there are only two basic kinds of expressions: atoms and lists (unstructured and structured). Lists are the only structure (a slight simplification -- but not by much).**

Scheme Sample Expressions

`42` —a number

`"hello"` —a string

`#T` —the Boolean value "true"

`#\a` —the character 'a'

`(2.1 2.2 3.1)` —a list of numbers

`a` —an identifier

`hello` —another identifier

`(+ 2 3)` —a list consisting of the identifier "+" and two numbers

`(* (+ 2 3) (/ 6 2))` —a list consisting of an identifier followed by two lists

Scheme Evaluation

- **Scheme programs are executed by evaluating expressions. A Scheme program consists of a series of expressions that are loaded and executed by the interpreter. Typically, all but the last of these expressions are definitions, and the last one (often entered directly) launches the program (much like `main` does in Java and C).**
- **The interpreter runs in a loop, called the "read-eval-print" loop: it reads an expression, evaluates it, and prints the result. Then it is ready to read a new expression. `Read`, `eval`, and `print` can also be called directly by programs. In fact, the interpreter is usually written in Scheme too (called a metacircular interpreter).**

Scheme evaluation rule

- **1. Constant atoms, such as numbers and strings, evaluate to themselves.**
- **2. Identifiers are looked up in the current environment and replaced by the value found there. (The environment in Scheme is essentially a dynamically maintained symbol table that associates identifiers to values.)**
- **3. A list is evaluated by recursively evaluating each element in the list as an expression (in some unspecified order); the first expression in the list must evaluate to a function. This function is then applied to the evaluated values of the rest of the list. Thus, all expressions are in *prefix form*.**

Comparison of C to Scheme

C	Scheme
`3 + 4 * 5`	`(+ 3 (* 4 5))`
`(a == b)` `  && (a != 0)`	`(and (= a b)` `  (not (= a 0)))`
`gcd(10,35)`	`(gcd 10 35)`
`gcd`	`gcd`
`getchar()`	`(read-char)`

Evaluation rule, continued

- **The Scheme evaluation rule is essentially the same as C or Java: arguments are evaluated at a call site before they are passed to the called function.**
- **This is usually called “applicative order evaluation”, or “eager” evaluation.**
- **Other kinds of evaluation *delay* the evaluation of the arguments.**

Examples of delayed evaluation in Scheme

- **The `if` function `(if a b c)`:**
  a is always evaluated, and, depending on whether the result is `#t` or `#f`, either b or c is evaluated and returned as the result.
- **The `define` function `(define x (+ 2 3))`:**
  this defines top-level names; the second expression is always evaluated, the first expression is never evaluated -- it must be a name whose value becomes the value of the second expression.
- **Such functions are called special forms or keywords.**

The `quote` special form

- **`Quote`, or `'` for short, has as its whole purpose to *not* evaluate its argument: `(quote (2 3 4))` returns just `(2 3 4)`.**
- **Another way to write this is `'(2 3 4)`.**
- **`Quote` is used to create data constants directly by "canceling" evaluation.**
- **We can in fact get evaluation back again: `(eval '(+ 2 3))` returns 5.**
- **More special forms: `cond` and `let`: similar to `switch` and `{...}` in Java and C.**

Scheme code examples

```
(define a 2)

(define emptylist '())

(define (gcd u v) ;defines a function
  (if (= v 0) u
      (gcd v (remainder u v))))

(cond((= a 0) 0) ;if (a==0) return 0;
     ((= a 1) 1) ;elsif(a==1)return 1;
     (else (/ 1 a))); else return 1/a;
```

Scheme examples, continued

```
; Figure 11.7, pages 486-487
(define (euclid)
  (display "enter two integers:")
  (newline)
  (let ((u (read)) (v (read)))
    (display "the gcd of ")
    (display u)
    (display " and ")
    (display v)
    (display " is ")
    (display (gcd u v))
    (newline)))
```

Special form syntax can be tricky

- **`if` is easy - it always takes three expressions as parameters: `(if a b c)`**
- **cond is not so easy - it takes an arbitrary sequence of expressions, each of which is a list of two (or more) items:**
  `(cond (e1 v1 …) (e2 v2 …) …)`
- **`define` has *two* forms -**
  `(define a b)`: a must be a name
  `(define (a …) b1 b2 …)`: defines function a with body the sequence b1, b2, etc.
- **`let` has implicit sequences too:**
  `(let ((n1 e1) (n2 e2) …) v1 v2 …)`

Lists in Scheme

- **Lists are (essentially) the only data structure in Scheme (like arrays in Algol60). However, lists are very versatile and can be used to model any other data structure.**
- **Lists are constructed recursively using the empty list `'()` and the `cons` binary operator:**
  `(1 2 3)` = `(cons 1 (cons 2 (cons 3 '()))`
- **The first element of a list (its head) is returned by the `car` operator: `(car '(1 2 3))` = `1`**
- **The tail of a list is returned by the `cdr` operator: `(cdr '(1 2 3))` = `(2 3)`**

List Examples:

cdr down

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

Predicate function

cons up

```
(define (reverse L)
  (if (null? L) '()
      (append (reverse (cdr L)) (list (car
L)))))
```

Creates a list out of a sequence

```
(define (square-list L)
  (if (null? L) '()
      (cons (* (car L) (car L)) (square-list
(cdr L)))))
```

Data structures in Scheme

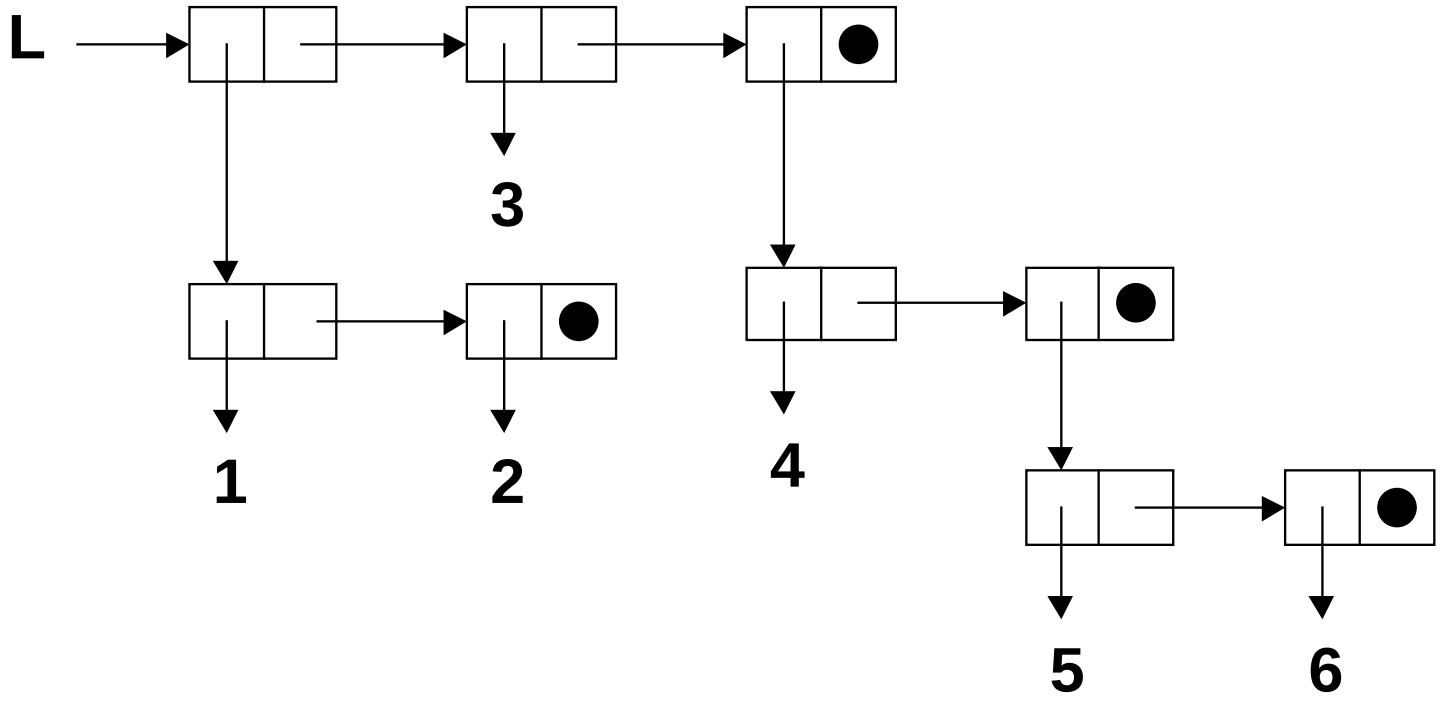
- **Since lists can recursively contain other lists, lists can model any recursive data structure:**

```
(define L '((1 2) 3 (4 (5 6))))
(car (car L)) => 1
(cdr (car L)) => (2)
(car (car (cdr (cdr L)))) => 4
```

- **Note:** `car(car = caar`
  `cdr(car = cdar`
  `car(car(cdr(cdr = caaddr`

Box diagrams: a visual aid

- **L = ((1 2) 3 (4 (5 6))) looks as follows in memory (notice how everything is a pointer):**



Binary search trees

- **Represent each node as a 3-item list (data left-child right-child):**

```
(define (data B) (car B))
(define (leftchild B) (cadr B))
(define (rightchild B) (caddr B))
```

- **Example - see Figure 11.8, page 487:**

```
("horse"("cow"()("dog"()()))
        ("zebra"("yak"()())()))
```

- **Now we can write traversals such as**

```
(define (tree-to-list B)
  (if (null? B) B
     (append (tree-to-list (leftchild B))
        (list (data B))
        (tree-to-list (rightchild B)))))
```

Equality in Scheme

Scheme has many different equality functions:

- **= applies only to numbers**
- **`char=?` applies only to characters**
- **`string=?` applies only to strings**
- **In general, every non-numeric data type has its own equality function**
- **There are also three "generic" equality functions: `eq?`, `eqv?`, and `equal?` The last function is the most general, testing structural equality similar to `Object.equal()` in Java; use this always for lists. The other two test identity in memory, similar to == applied to objects in Java.**

Equality of functions in Scheme

- **Testing functions for equality is tricky:**

```
(define (f x) x)
(define (g x) x) ; clearly g = f
(define (h y) (car (list y)))
; now h = f too, but how to tell?
```

- **Thus, most Scheme versions decide that functions are *never* equal unless they occupy the same memory:**

```
(equal? f g) => #f
(define k f)
(equal? f k) => #t
```

`Eval` and symbols

- **The `eval` function evaluates an expression in an environment; many Scheme versions have an implied current environment:**
  `(eval (cons max '(1 3 2)) => 3`
- **Symbols are virtually unique to Lisp: they are runtime variable names, or unevaluated identifiers:**
  `'x => x`
  `(eval 'x)` => the value of `x` in the environment
- **Use symbols for enums (they are more efficient than strings)**

Lambda expressions/function values

- **A function can be created dynamically using a `lambda` expression, which returns a value that is a function (aka procedure):**

```
> (lambda (x) x)
#<procedure>
```

- **The syntax of a `lambda` expression is (lambda *list-of-parameters exp1 exp2* …)**
- **Indeed, the "function" form of `define` is just syntactic sugar for a `lambda`:**

```
(define (f x) x)
```

is equivalent to:

```
(define f (lambda (x) x))
```

Function values as data

- **The result of a `lambda` can be manipulated as ordinary data:**

```
> ((lambda (x) (* x x)) 5)
25
(define f (list (lambda () 2)))
> f
(#<procedure>)
> (eval f)
2
> (define (add-x x) (lambda(y)(+ x y)))
> (define add-2 (add-x 2))
> (add-2 15)
17
```

Higher-order functions

- **Any function which returns a function as its value, or takes a function as a parameter, or both is called "higher-order"**
- **The `add-x` function of the previous slide is higher-order**
- **`Eval` is higher-order**
- **The use of higher-order functions is characteristic of functional programs**

Higher-order examples

```
(define (compose f g)
  (lambda (x) (f (g x))))
(define (map f L)
  (if (null? L) L
      (cons (f (car L))(map f (cdr L)))))
(define (filter p L)
  (cond
    ((null? L) L)
    ((p (car L)) (cons (car L)
                       (filter p (cdr L))))
    (else (filter p (cdr L)))))
```

`Let` expressions as lambdas:

- **A `let` expression is really just a lambda applied immediately:**
  ```
  (let ((x 2) (y 3)) (+ x y))
  ```
  is the same as
  ```
  ((lambda (x y) (+ x y)) 2 3)
  ```
- **This is why the following `let` expression is an error if we want x = 2 throughout:**
  ```
  (let ((x 2) (y (+ x 1))) (+ x y))
  ```
- **To compensate, there are sequential and recursive `lets`: `let*` and `letrec`**

Functions and objects

- **Functions can be used to model objects and classes in Scheme.**
- **Consider the simple Java class:**

```
public class BankAccount
{ public BankAccount(double initialBalance)
   { balance = initialBalance; }
   public void deposit(double amount)
   { balance = balance + amount; }
   public void withdraw(double amount)
   { balance = balance - amount; }
   public double getBalance()
   { return balance; }
   private double balance;
}
```

Functions and objects (cont'd)

- **This can be modeled in Scheme as:**

```
(define (BankAccount balance)
  (define (getBalance) balance)
  (define (deposit amt)
    (set! balance (+ balance amt)))
  (define (withdraw amt)
    (set! balance (- balance amt)))
  (lambda (message)
    (cond
      ((eq? message 'getBalance) getBalance)
      ((eq? message 'deposit) deposit)
      ((eq? message 'withdraw) withdraw)
      (else (error "unknown message" message)))))
```

Functions and objects (cont'd)

- **This code can be used as follows:**

```
> (define acct1 (BankAccount 50))
> (define acct2 (BankAccount 100))
> ((acct1 'getbalance))
50
> ((acct2 'getbalance))
100
> ((acct1 'withdraw) 40)
> ((acct2 'deposit) 50)
> ((acct1 'getbalance))
10
> ((acct2 'getbalance))
150
> ((acct1 'setbalance) 100)
. unknown message setbalance
```

Functions and objects (cont'd)

- **Note the use of the assignment operator `set!` in this code. Thus, this code is definitely not purely functional.**
- **In Scheme, any function ending in "!" is a non-functional, imperative-style operation. There are several of these: `set!`, `set-car!`, `set-cdr!`, `string-set!`, etc. These are all different versions of assignment.**
- **For a Scheme program to be purely functional, there should be no use of these functions.**