Functional Programming

Applications of functional programming

- Prototyping
- Artificial intelligence
- Mathematical proof systems
- Logic applications

Advantages of functional programming

- No side effects
- Automatic memory management
- Great flexibility
- Simple semantics
- Uniform view of programs as functions
- The treatment of functions as data
- Conciseness of notation

Major drawbacks of functional programming

- Efficiency is less
- Mathematical in nature
- Abstract in nature

Programs as functions

- A program is a description of a specific computation.
- If we ignore the details of the computation—the "how" of the computation—and focus on the result being computed—the "what" of the computation, then a program becomes a virtual black box that transforms input into output.
- From this point of view, a program is essentially equivalent to a mathematical function.
- Definition: A *function is a rule that associates to each x from some set X of values a unique y* from a set *Y of values. In mathematical terminology, if f is the name of the function, we write:*

$$y=f(x) \quad \text{or} \quad f: X \rightarrow Y$$

The set X is called domain of f.

The set Y is called range of f.

x is called independent variable.

y is called dependent variable.

Partial and Total functions

- If f is not defined for all x in X then f is called partial function.
- If f is defined for all x in X then f is called total function.

Programs as functions

- Programs, procedures, and functions in a programming language can all be represented by the mathematical concept of a function.
- In the case of a program, *x represents the input and y represents the* output.
- In the case of a procedure or function, *x represents the parameters and y represents the returned* values.
- In either case, we can refer to *x as "input" and y as "output."*
- *Thus, the functional view of programming* makes no distinction between a program, a procedure, and a function. It always makes a distinction, however, between input and output values.

No Assignment

- In Mathematics, variables always stand for actual values, while in imperative programming languages, variables refer to memory locations that store values and assignment allows these memory locations to be reset with new values.
- In mathematics, there are no concepts of memory location and assignment, so that a statement such as $x = x + 1$ makes no sense.
- Functional programming takes a mathematical approach to the concept of a variable.
- In functional programming, variables are bound to values, not to memory locations. Once a variable is bound to a value, that variable's value cannot change.
- This also eliminates assignment, which can reset the value of a variable, as an available operation.

No loops

- One consequence of lack of assignment in functional programming is that there can be no loops.
- A loop must have a control variable that is reassigned as the loop executes and this is not possible without the assignment.
- Functional languages have recursion instead of loops.

Euclid Algorithm

```
void gcd( int u, int v, int* x)
{       int y, t, z;
        z = u ; y = v;
        while (y != 0)
        {
                t = y;
                y = z % y;
                z = t;
        }
        *x = z;
}
```

Imperative version using a loop

```
int gcd(int u, int v)
{
    if(v==0) return u;
     else return gcd(v,u%v);
}
```

Functional version with recursion

Mathematical definition of GCD

- The second form is close to the mathematical (that is, recursive) definition of the function as

$$\gcd(u, v) = \begin{cases} u & \text{if } v = 0 \\ \gcd(v, u \bmod v) & \text{otherwise} \end{cases}$$

Referential Transparency

- The property of a function that its value depends only on the values of its arguments is called **referential transparency**.
- GCD function is referentially transparent because its value depends only on the value of its arguments.
- Which function is not referentially transparent in programming languages?
- On the other hand, a function rand, which returns a pseudo random value, cannot be referentially transparent, because it depends on the state of the machine (and previous calls to itself).

Value semantics

- The runtime environment associates names to values only (not memory locations) and once a name enters the environment its value can never change, such a notion of semantic is called as value semantics.

First-class data values

- Functions must be viewed as values themselves, which can be computed by other functions and which can also be parameters to functions. Thus we can say in functional programming, functions are **first-class data values**.

Composition

- One of the essential operation on functions is composition.
- Composition is itself a function that takes two functions as parameters and produces another function as its returned value.
- Functions like this—that have parameters that are themselves functions or that produce a result that is a function, or both—are called **higher-order functions**
- Mathematically, the composition operator ‘o’ is defined as follows if f:X→Y and g:Y→Z then gof:X→Z is given by (gof)(X)= g(f(X))

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;
}
```

Tail recursion

- A recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.

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);}
```

Summary of qualities of Functional Programming

1. All procedures are functions and clearly distinguish incoming values (parameters) from outgoing values (results).
2. In pure functional programming, there are no assignments. Once a variable is bound to a value, it behaves like a constant.
3. In pure functional programming, there are no loops. Instead, loops are replaced by recursive calls.
4. The value of a function depends only on the value of its parameters and not on the order of evaluation or the execution path that led to the call.
5. Functions are first-class data values.

Scheme

- Scheme is variation of LISP.

Elements of Scheme

- All programs and data in scheme are expressions.
- There are two types of expressions: atoms and parenthesized sequences of expressions.
- Atoms are like the literal constants and identifiers of an imperative language: they include numbers, strings, names and functions.
- A parenthesized expression is simply a sequence of zero or more expressions separated by spaces and surrounded by parentheses.
- Thus, the syntax of Scheme is particularly simple:

  expression → atom | '(' {expression} ')'

  atom → number | string | symbol | character | boolean

Scheme Syntax

- This syntax is expressed in a notation called **extended Backus-Naur form.**

Symbols used in an extended Backus-Naur form grammar

Symbol	Use
→	Means "is defined as"
|	Indicates an alternative
{ }	Enclose an item that may be seen zero or more times
' '	Enclose a literal item

Scheme Expression

- When parenthesized expressions are viewed as data, we call them lists.
- Some examples of Scheme expressions are shown in Table

Some Scheme expressions

Expression	What It Represents
42	an integer value
"hello"	a string
#T	the Boolean value "true"
#\a	the character 'a'
(2.1 2.2 3.1)	a list of real numbers
a	a symbol
hello	another symbol
(+ 2 3)	a list consisting of the symbol + and two integers
(* (+ 2 3) (/ 6 2))	a list consisting of a symbol followed by two lists

Evaluation rule for expression

- The meaning of a Scheme expression is given by an **evaluation rule.**
- The standard evaluation rule for Scheme expressions is as follows:
  1. Atomic literals, such as numbers, characters, and strings, evaluate to themselves.
  2. Symbols other than keywords are treated as identifiers or variables that are looked up in the current environment and replaced by the values found there.

Evaluation rule for expression

- A parenthesized expression or list is evaluated in one of two ways:

  a. If the first item in the expression is a Scheme keyword, such as if or let, a special rule is applied to evaluate the rest of the expression. Scheme expressions that begin with keywords are called **special forms**, and are used to control the flow of execution.

  b. Otherwise, the parenthesized expression is a function application. Each expression within the parentheses is evaluated recursively in some unspecified order; the first expression among them must evaluate to a function. This function is then applied to remaining values, which are its arguments.

Value of function V/S function call

- The Scheme evaluation rule implies that all expressions in Scheme must be written in prefix form.
- It also implies that the value of a function (as an object) is clearly distinguished from a call to the function:
- The function value is represented by the first expression in an application, while a function call is surrounded by parentheses.
- Thus, a Scheme interpreter would show behavior similar to the following:

```
> +
#[PROCEDURE: +]
> (+ 2 3)
5 ; a call to the + procedure with arguments 2 and 3
```

- + and (+) → value of function vs function call
- Note that a Scheme end-of-line comment begins with a semicolon.

Comparison of C and Scheme

- 3+4*5
- (a==b) && (a!=0)
- gcd(10,35)
- gcd
- getchar()

- (+ 3(* 4 5))
- (and(= a b)(not (= a 0)))
- (gcd 10 35)
- gcd
- (read-char)

Applicative order evaluation

- All sub expressions are evaluated first so that the expression tree is evaluated from leaves to the root.
- (*(+ 2 3)(+ 4 5))➔(* 5 9)➔45

Quote function

- Quote is a special function whose sole purpose is to stop evaluation.

>(2.1 2.2 2.3)

Error: the object 2.1 is not a procedure

>(quote (2.1 2.2 2.3))

(2.1 2.2 2.3)

> (quote scheme)

Scheme (expression returns a Scheme **symbol**)

>’(2.1 2.2 2.3) //The quote special form is used so often that there is a special shorthand notation for it—the single quotation mark or apostrophe:

(2.1 2.2 2.3)

If function

- Loops are provided by recursive call, but selection must be given by special forms.
- The basic forms that do this are the if form, which is like an if-else construct, and the cond form, which is like an if-elsif construct. (Note that cond stands for **conditional expression.)**

- (if (= a 0) 0 (/ 1 a)); if a=0 then return 0
  ;else return 1/a

Cond function

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

if function

- The semantics of the expression (if *exp1 exp2 exp3) dictate that exp1 is evaluated first; if the* value of *exp1 is the Boolean value false (#f), then exp3 is evaluated and its value returned by the if* expression; otherwise, *exp2 is evaluated and its value is returned.*
- *Also, exp3 may be absent; in that case,* if *exp1 evaluates to false, the value of the expression is undefined.*

Cond function

- Similarly, the semantics of (cond *exp1 . . . expn) dictate that each expi must be a pair of expressions:*
- *expi = (fst snd). Each expression expi is considered in order, and the first part of it is evaluated. If fst evaluates to #T (true), then snd is evaluated, and its value is returned by the cond expression. If none* of the conditions evaluates to #T, then the expression in the else clause is evaluated and returned as the value of the cond expression (i.e., the keyword else in a cond can be thought of as always evaluating to #T).
- If there is no else clause (the else clause is optional) and none of the conditions evaluates to #T, then the result of the cond is undefined.

let function

- Another important special form is the let, which allows variable names to be bound to values within an expression:

```
> (let ((a 2) (b 3)) (+ a b))
5
```

- The first expression in a let is a **binding list,** which associates names to values.
- In this binding list of two bindings, a is given the value 2 and b the value 3.
- The names a and b can then be used for their values in the second expression, which is the body of the let and whose value is the returned value of the let.
- Note that let provides a local environment and scope for a set of variable names, in much the same manner as temporary variable declarations in block-structured languages such as Java and Python.
- The values of these variables can be accessed within the let form, but not outside it.

Declaring functions

- (define a 2)
- (define emptylist ‘())
- (define (gcd u v)
  (if(= v 0) u (gcd v(remainder u v))))

Command prompt

```
>a
   2
>emptylist
   ()
>gcd
   #[procedure: GCD]
>(gcd 25 10)
   5
```

Read

- read function:- The read function has no parameters. It returns whatever value the keyboard provides.

```
>read
#procedure READ
>(read)
  234
234
```

Display function

- The display function prints its parameters to the screen.

  >(display 234)

  234

Euclids GCD Interactive

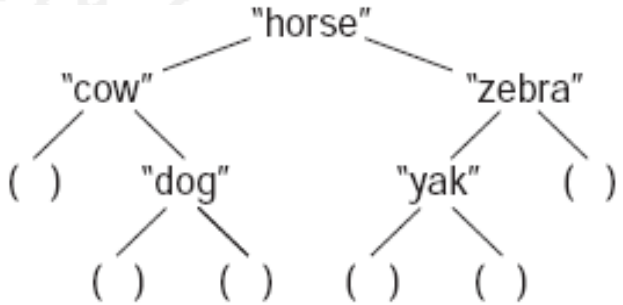
```
(define (euclid)
(display “enter two integers”)
(newline)
(let ((u (read))(v (read))
(display “the gcd of”)
(display u)
(display “and”)
(display v)
(display “is”)
(display (gcd u v))
(newline)))
```

Data Structures in scheme

- Basic data structure in scheme is the list.
- Although Scheme also supports structured types for vectors (one-dimensional arrays) and strings, a list can represent a sequence, a record, or any other structure.
- For example, the following is a list representation of a binary search tree:

  ("horse" ("cow" () ("dog" () ()))

  ("zebra" ("yak" () ()) ()))
- A node in this tree is a list of three items (name left right), where name is a string, and left and right are the child trees, which are also lists. Thus, the given list represents the tree of Figure



Basic functions

- selector functions car and cdr, which access the head and the tail of a list
- car: compute head of a list
- cdr: compute tail of a list.

- the constructor function cons
- cons: adds a new head to an existing list.

Example

- If L is the list (1 2 3) then

  >(car L)

  1

  >(cdr L)

  >(2 3)

  (cons 4 L)

  (4 1 2 3)

CAR and CDR

- car: contents of the address register.
- cdr: contents of the decrement register.

shortcuts

- On account of the single letter difference in the names of the operations, repeated applications of both can be combined by combining the letters “a” and “d” between the “c” and the “r.” Thus,

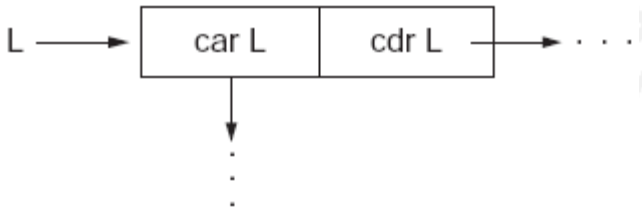
  (car(cdr L))➔(cadr L)

  (cdr(cdr L))➔(cddr L)

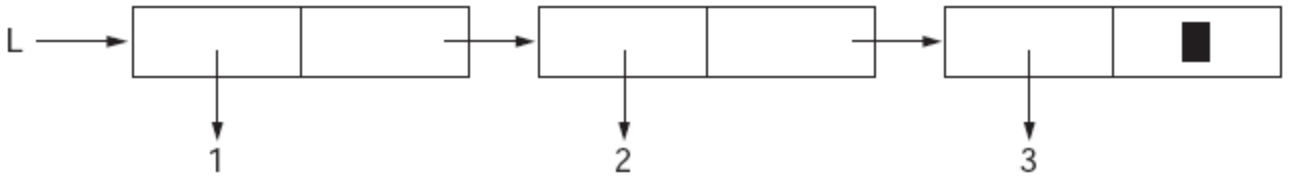
  (car(cdr(cdr L))) ➔(caddr L)

Representation or Visualization of List

- The view of a list as a pair of values represented by the car and the cdr has also continued to be useful as a representation or visualization of a list.
- According to this view, a list L is a pointer to a “box” of two pointers, one to its car and the other to its cdr.



- This box and pointer notation for a simple list such as (1 2 3) is shown in Figure



- The symbol black rectangle in the box at the end stands for the empty list ().

List Examples:

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

cdr down

Predicate function

cons up

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

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

Creates a list out of a sequence

All the basic list manipulation operations can be written as functions using the primitives car, cdr, cons, and null?. The last function is a predicate that returns true if its list argument is empty, or false otherwise.

Data structures in Scheme

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

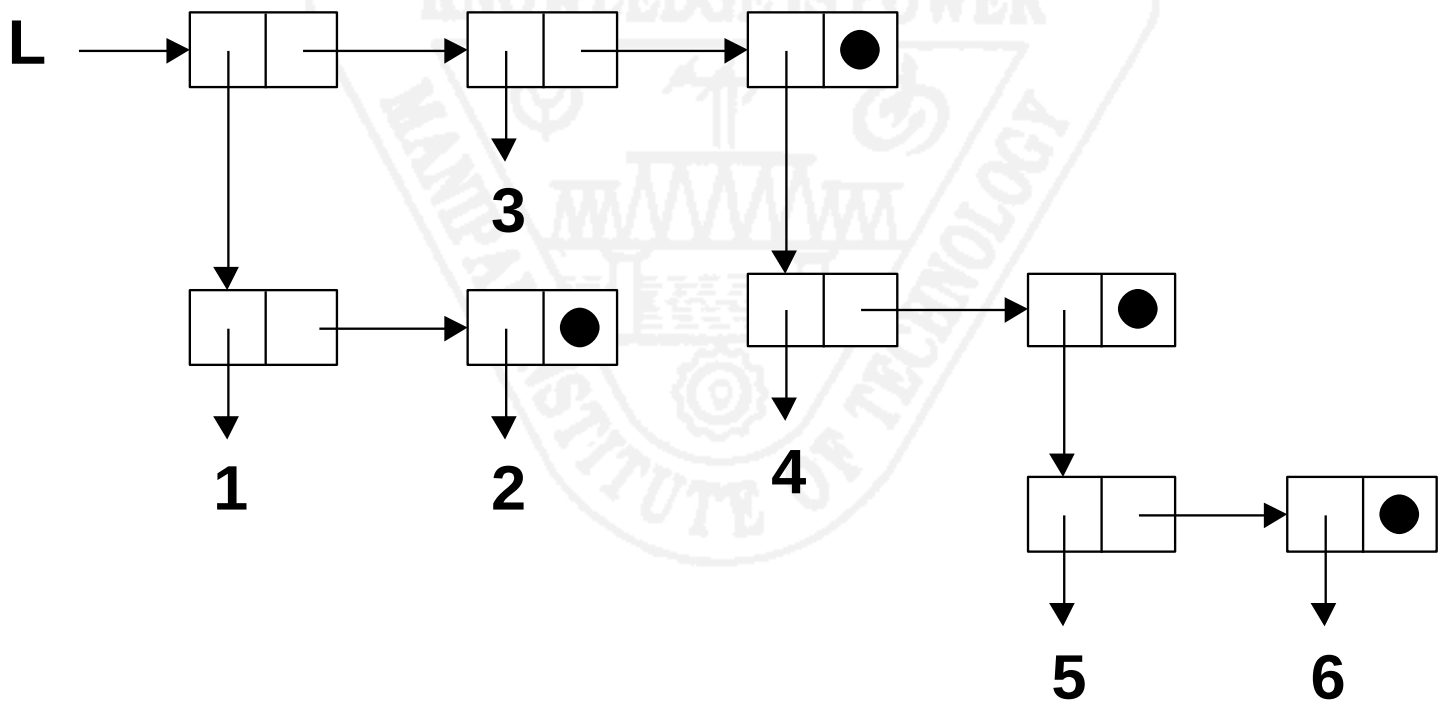
```
(define L '((1 2) 3 (4 (5 6))))
(car (car L)) => 1
(cdr (car L)) => (2)
(car (car (cdr (cdr L)))) => 4
```

- 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):



Draw the box diagram for the following

- ((((a))))
- (1 (2 (3 4)) (5))
- ((a ()) ((c) (d) b) e)

Extract the following

- a of the first list
- 3 of the second list
- c of the third list

Value of following

- (car (car L))
- (car (cdr L))
- (car (cdr (cdr (cdr L))))
- (cdr (car (cdr (cdr L))))

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)))))
```

Write scheme programs for the following

- Factorial
- Fibonacci
- Length of the list

Factorial

- (define (factorial N)
- (if (= N 0) 1
- (* N (factorial (- N 1)))))

Fibonacci

- (define (fibo N)
- (if (= N 0) 0
- (if (= N 1) 1
- (+ (fibo (- N 1)) (fibo (- N 2))))))

Length of the list

```
(define (count-elements L)
  (if (null? L) 0
      (+  1 (count-elements (cdr L)))))
```

Tail recursive length of the list

```
(define L `(1 2 3 4 5))
(define N 0)
(define (length-list-tail L N)
  (if (null? L) N
      (length-list-tail (cdr L) (+ N 1))))
```

References

Text book

- Kenneth C. Louden “Programming Languages Principles and Practice” second edition Thomson Brooks/Cole Publication.

Reference Books:

- Terrence W. Pratt, Masvin V. Zelkowitz “Programming Languages design and Implementation” Fourth Edition Pearson Education.
- Allen Tucker, Robert Noonan “Programming Languages Principles and Paradigms second edition Tata MC Graw –Hill Publication.