Module 5

Functional Programming Languages

- The functional programming paradigm, which is based on mathematical functions, is the design basis of the most important nonimperative styles of languages. This style of programming is supported by functional programming languages.
- Eg: LISP,Scheme

Mathematical Functions

- A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set.
- Function definition specifies the domain and range sets, either explicitly or implicitly, along with the mapping.The mapping is described by an expression.

Simple Functions

- Function definitions are often written as a function name, followed by a list of parameters in parentheses, followed by the mapping expression. For example,

  cube(x) ≡ x * x * x, where x is a real number

- The symbol ≡ is used to mean “is defined as.” The parameter x can represent any member of the domain set

lambda expression

- Lambda notation, as devised by Alonzo Church (1941), provides a method for defining nameless functions.
- A **lambda expression** specifies the parameters and the mapping of a function. The lambda expression is the function itself, which is nameless. For example, consider the following lambda expression:

  λ(x)x * x * x
- Church defined a formal computation model (a formal system for function definition, function application, and recursion) using lambda expressions. This is called **lambda calculus**.
- Lambda calculus can be either typed or untyped.
- Untyped lambda calculus serves as the inspiration for the functional programming languages.
- example lambda expression is denoted as in the following example:

  (λ(x)x * x * x)(2) which results in the value 8.
- Lambda expressions, like other function definitions, can have more than one parameter.

Functional Forms

- A **higher-order function**, or **functional form**, is one that either takes one or more functions as parameters or yields a function as its result, or both.
- One common kind of functional form is **function composition**, which has two functional parameters and yields a function whose value is the first actual parameter function applied to the result of the second.
- Function composition is written as an expression, using ° as an operator, as in $h \equiv f \circ g$

- For example, if

  $f(x) \equiv x + 2$

  $g(x) \equiv 3 * x$

- then h is defined as $h(x) \equiv f(g(x))$, or $h(x) \equiv (3 * x) + 2$

- **Apply-to-all** is a functional form that takes a single function as a parameter**.** Apply-to-all is denoted by α . Consider the following example:

- Let $h(x) \equiv x * x$ then $\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$

The First Functional Programming Language: LISP

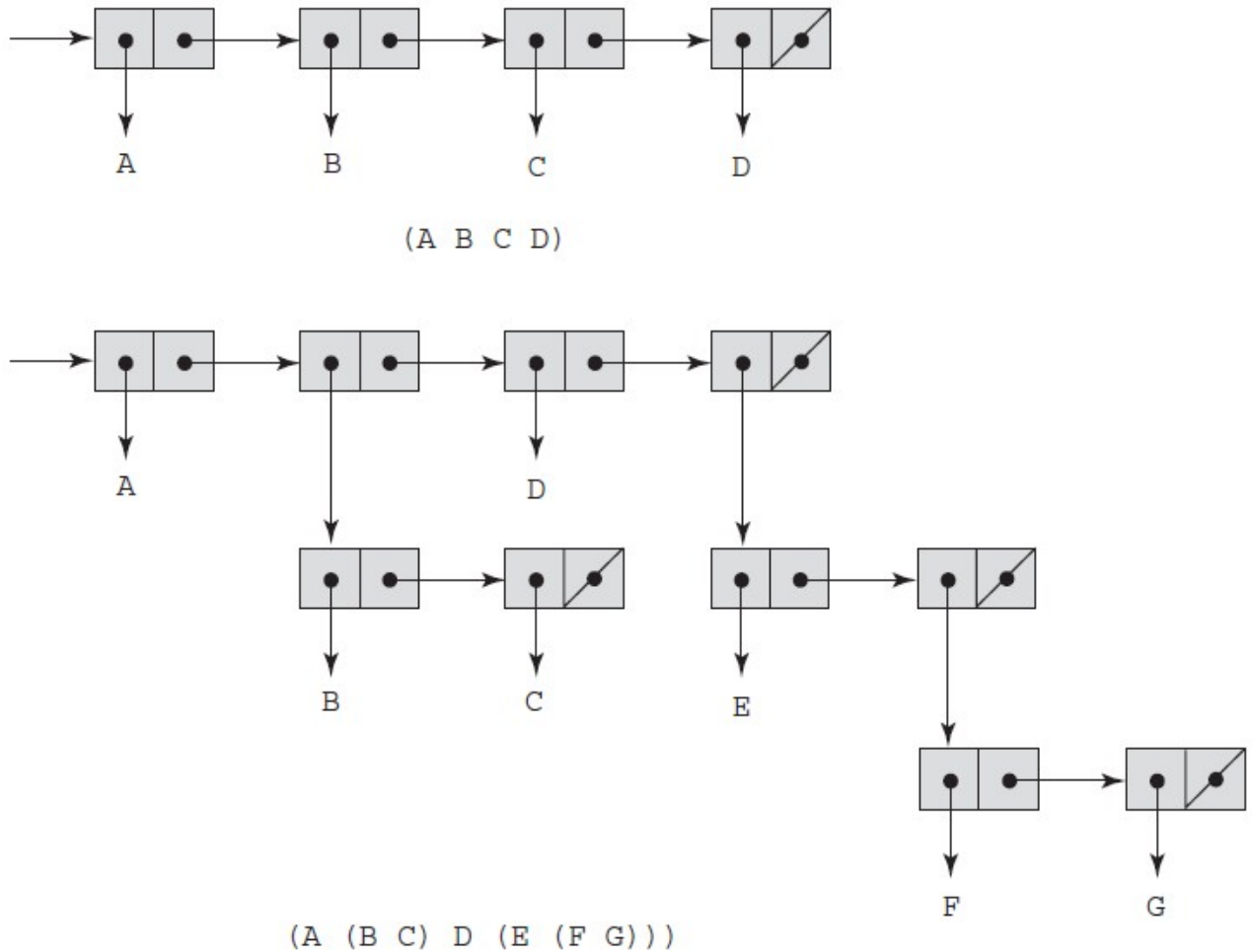
- LISP was the first functional language.

Data Types and Structures

- There were only two categories of data objects in the original LISP: atoms and lists.
- List elements are pairs, where the first part is the data of the element,which is a pointer to either an atom or a nested list. The second part of a pair can be a pointer to an atom, a pointer to another element, or the empty list.
- the original LISP was a typeless language. Atoms are either symbols, in the form of identifiers,or numeric literals.
- The elements of **simple lists** are restricted to atoms, as in
- `(A B C D)`
- Nested list structures are also specified by parentheses. For example,
- the list `(A (B C) D (E (F G)))`
- is a list of four elements. The first is the atom `A`; the second is the sublist `(B C)`; the third is the atom `D`; the fourth is the sublist `(E (F G))`, which has as its second element the sublist `(F G)`.

Figure 15.1

Internal representation of two LISP lists



The First LISP Interpreter

- The first requirement for the universal LISP function was a notation that allowed functions to be expressed in the same way data was expressed.
- Function calls were specified in a prefix list form originally called **Cambridge Polish**, as in the following:

  (function_name argument1 …….argumentn)

- For example, if + is a function that takes two or more numeric parameters, the following two expressions evaluate to `12` and `20`, respectively:

```
(+ 5 7)

(+ 3 4 7 6)
```

- LISP functions specified in new notation were called S-expressions, for *s*ymbolic expressions. All LISP structures, both data and code, were called S-expressions. An S-expression can be either a list or an atom.

An Introduction to Scheme

The Scheme Interpreter

- A Scheme interpreter in interactive mode is an infinite read-evaluate-print loop (often abbreviated as REPL).
- It repeatedly reads an expression typed by the user (in the form of a list), interprets the expression, and displays the resulting value. This form of interpreter is also used by Ruby and Python. Expressions are interpreted by the function `EVAL`.
- Expressions that are calls to primitive functions are evaluated in the following way: First, each of the parameter expressions is evaluated, in no particular order. Then, the primitive function is applied to the parameter values, and the resulting value is displayed.
- Scheme programs that are stored in files can be loaded and interpreted.
- Comments in Scheme are any text following a semicolon on any line.

Primitive Numeric Functions

- Scheme includes primitive functions for the basic arithmetic operations. These are `+`, `-`, `*`, and `/`, for add, subtract, multiply, and divide. `*` and `+` can have zero or more parameters. If `*` is given no parameters, it returns `1`; if `+` is given no parameters, it returns `0`. `+` adds all of its parameters together. `*` multiplies all its parameters together. `/` and `-` can have two or more parameters.

Expression	*Value*
`42`	`42`
`(* 3 7)`	`21`
`(+ 5 7 8)`	`20`
`(- 5 6)`	`-1`
`(- 15 7 2)`	`6`

- There are a large number of other numeric functions in Scheme, among them `MODULO`, `ROUND`, `MAX`, `MIN`, `LOG`, `SIN`, and `SQRT`. `SQRT` returns the square root of its numeric parameter, if the parameter's value is not negative.

Defining Functions

- A Scheme program is a collection of function definitions.
- In Scheme, a nameless function actually includes the word `LAMBDA`, and is called a **lambda expression**. For example,

  `(LAMBDA (x) (* x x))` is a nameless function that returns the square of its given numeric parameter.

- This function can be applied in the same way that named functions are: by placing it in the beginning of a list that contains the actual parameters.
- For example, the following expression yields `49`: `((LAMBDA (x) (* x x)) 7)`
- In this expression, `x` is called a **bound variable** within the lambda expression. During the evaluation of this expression, `x` is bound to `7`.
- Lambda expressions can have any number of parameters. For example,

```
(LAMBDA (a b c x) (+ (* a x x) (* b x) c))
```

- `DEFINE` is called a special form because it is interpreted (by `EVAL`) in a different way than the normal primitives like the arithmetic functions.
- The simplest form of `DEFINE` is one used to bind a name to the value of an expression. This form is

  `(DEFINE` symbol expression`)`
- For example,

```
(DEFINE pi 3.14159)
(DEFINE two_pi (* 2 pi))
```

- `Another use`

  `(DEFINE (`function_name parameters`)`

  `(`expression`)`

  `)`
- this form of `DEFINE` is simply the definition of a named function.
- The following example call to `DEFINE` binds the name `square` to a functional expression that takes one parameter:

```
(DEFINE (square number) (* number number))
```

- After the interpreter evaluates this function, it can be used, as in `(square 5)` which displays `25`.

Output Functions

- Scheme includes a formatted output function, `PRINTF`, which is similar to the `printf` function of C.

Numeric Predicate Functions

- A predicate function is one that returns a Boolean value (some representation of either true or false). Scheme includes a collection of predicate functions for numeric data. Amc

Function	*Meaning*
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
EVEN?	Is it an even number?
ODD?	Is it an odd number?
ZERO?	Is it zero?

In Scheme, the two Boolean values are `#T` and `#F`

Control Flow

- The Scheme two-way selector function, named `IF`, has three parameters: a predicate expression, a then expression, and an else expression. A call to `IF` has the form

  `(IF` predicate then_expression else_expression`)`

- For example,

```
(DEFINE (factorial n)
   (IF (<= n 1)
   1
   (* n (factorial (- n 1)))
   ))
```

List Functions

- the primitive function `QUOTE`, which simply returns it without change.
- For eg

  `(QUOTE A)` returns `A`

  `(QUOTE (A B C))` returns `(A B C)`

- operations of `CAR` and `CDR`:
  - `(CAR '(A B C))` returns `A`
  - `(CAR '((A B) C D))` returns `(A B)`
  - `(CAR 'A)` is an error because `A` is not a list
  - `(CAR '(A))` returns `A`
  - `(CAR '())` is an error
  - `(CDR '(A B C))` returns `(B C)`
  - `(CDR '((A B) C D))` returns `(C D)`
  - `(CDR 'A)` is an error
  - `(CDR '(A))` returns `()`
  - `(CDR '())` is an error
- Some of the most commonly used functional compositions in Scheme are built in as single functions. For example, `(CAAR x)` is equivalent to `(CAR(CARx))`, `(CADR x)` is equivalent to `(CAR (CDR x))`, and `(CADDAR x)` is equivalent to `(CAR (CDR (CDR (CAR x))))`.
- consider the following evaluation of `CADDAR`:

```
(CADDAR '((A B (C) D) E)) =

(CAR (CDR (CDR (CAR '((A B (C) D) E))))) =

(CAR (CDR (CDR '(A B (C) D)))) =

(CAR (CDR '(B (C) D))) =

(CAR '((C) D)) =

(C)
```

- Following are example calls to `CONS`:

`(CONS 'A '())` returns `(A)`

`(CONS 'A '(B C))` returns `(A B C)`

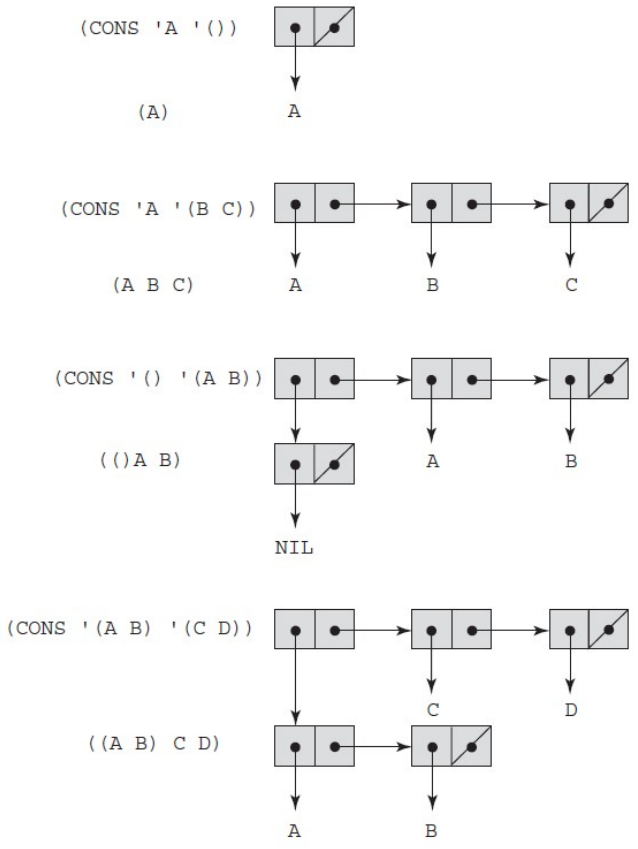
`(CONS '() '(A B))` returns `(() A B)`

`(CONS '(A B) '(C D))` returns `((A B) C D)`

- `CONS` is, the inverse of `CAR` and `CDR`. `CAR` and `CDR` take a list apart, and `CONS` constructs a new list from given list parts. The two parameters to `CONS` become the `CAR` and `CDR` of the new list.

Figure 15.2

The result of several CONS operations



- `(CONS 'A 'B)` If the result of this is displayed, it would appear as `(A . B)`
- `LIST` is a function that constructs a list from a variable number of parameters.

  `(LIST 'apple 'orange 'grape)` returns `(apple orange grape)`

Predicate Functions for Symbolic Atoms and Lists

- Scheme has three fundamental predicate functions, `EQ?`, `NULL?`, and `LIST?`
- The `EQ?` function takes two expressions as parameters, although it is usually used with two symbolic atom parameters. It returns `#T` if both parameters have the same pointer value—that is, they point to the same atom or list; otherwise, it returns `#F`

  `(EQ? 'A 'A)` returns `#T`

  `(EQ? 'A 'B)` returns `#F`

  `(EQ? 'A '(A B))` returns `#F`

  `(EQ? '(A B) '(A B))` returns `#F` or `#T`

  `(EQ? 3.4 (+ 3 0.4))` returns `#F` or `#T`

- `EQV?`, which works on both numeric and symbolic atoms. Consider the following examples:

  `(EQV? 'A 'A)` returns `#T`

  `(EQV? 'A 'B)` returns `#F`

  `(EQV? 3 3)` returns `#T`

  `(EQV? 'A 3)` returns `#F`

  `(EQV? 3.4 (+ 3 0.4))` returns `#T`

  `(EQV? 3.0 3)` returns `#F`

- The `LIST?` predicate function returns `#T` if its single argument is a list and `#F` otherwise, as in the following examples:

  `(LIST? '(X Y))` returns `#T`

  `(LIST? 'X)` returns `#F`

  `(LIST? '())` returns `#T`

- The `NULL?` function tests its parameter to determine whether it is the empty list and returns `#T` if it is. Consider the following examples:

  `(NULL? '(A B))` returns `#F`

  `(NULL? '())` returns `#T`

  `(NULL? 'A)` returns `#F`

  `(NULL? '(()))` returns `#F`

LET

- `LET` is a function that creates a local scope in which names are temporarily bound to the values of expressions.
- It is often used to factor out the common subexpressions from more complicated expressions.
- These names can then be used in the evaluation of another expression, but they cannot be rebound to new values in `LET`.
- roots of the quadratic equation ax2 + bx + c are as follows: root1 = (-b + sqrt(b2 - 4ac))/2a and root2 = (-b - sqrt(b2 - 4ac))/2a

```
(DEFINE (quadratic_roots a b c)
(LET (
(root_part_over_2a
(/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
(minus_b_over_2a (/ (- 0 b) (* 2 a)))
)
(LIST (+ minus_b_over_2a root_part_over_2a)
(- minus_b_over_2a root_part_over_2a))
))
```

- `LET` is actually shorthand for a `LAMBDA` expression applied to a parameter.
- The following two expressions are equivalent:

```
(LET ((alpha 7))(* 5 alpha))

((LAMBDA (alpha) (* 5 alpha)) 7)
```

- In the first expression, `7` is bound to `alpha` with `LET`; in the second, `7` is bound to `alpha` through the parameter of the `LAMBDA` expression.

Tail Recursion in Scheme

- A function is **tail recursive** if its recursive call is the last operation in the function.
- This means that the return value of the recursive call is the return value of the nonrecursive call to the function.

```
(DEFINE (member atm a_list)
(COND
((NULL? a_list) #F)
((EQ? atm (CAR a_list)) #T)
(ELSE (member atm (CDR a_list)))
))
```

- Programmers who were concerned with efficiency have discovered ways to rewrite some of these functions so that they are tail recursive. One example of this uses an accumulating parameter and a helper function.
- consider the factorial function

```
(DEFINE (factorial n)
   (IF (<= n 1)
   1
   (* n (factorial (- n 1)))
   ))
```

- The last operation of this function is the multiplication. This factorial function can be rewritten with an auxiliary helper function, which uses a parameter to accumulate the partial factorial. The helper function, which is tail recursive, also takes `factorial`'s parameter.

```
(DEFINE (facthelper n factpartial)
(IF (<= n 1)
factpartial
(facthelper (- n 1) (* n factpartial))
))
(DEFINE (factorial n)
(facthelper n 1)
)
```

Functional Forms

- two common mathematical functional forms that are provided by Scheme: composition and apply-to-all

Functional Composition

- Functional composition is the only primitive functional form provided by the original LISP.
- function composition is a functional form that takes two functions as parameters and returns a function that 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))`.
- For example,

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
```

- Now the functional composition of `f` and `g` can be written as follows:

```
(DEFINE (h x) (+ 2 (* 3 x)))
```

- In Scheme, the functional composition function `compose` can be written 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`.
- `((compose CDR CAR) '((a b) c d))` This call would yield `(b)`.

An Apply-to-All Functional Form

- `map`, which has two parameters: a function and a list. `map` applies the given function to each element of the given list and returns a list of the results of these applications.
- suppose we want all of the elements of a list cubed. We can accomplish this with the following:

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

- This call returns `(27 64 8 216)`.