Chapter 15 Functional Programming Languages

- Fundamentals of Functional Programming Languages
- Introduction to Scheme

A programming paradigm treats computation as the evaluation of mathematical functions. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state.

Thus, programs are collections of mathematical function definitions and function applications.

- It emphasized in academia rather than in commercial software development. Languages used in commercial applications include Mathematica (symbolic math), Haskell, ML (financial analysis), and XSLT.
- It includes APL, Haskell, Lisp, ML, Scheme, XSLT, etc.

What is a mathematical function?

A rule that associates members of one set (the domain) with members of another set.

```
cube(x) ≡ x * x * x
```

Early theoretical work on mathematical functions separated the task of defining a function from that of naming the function.

Lambda expression provides a method for defining **nameless** functions.

Lambda Expressions

Syntax: `LAMBDA` *(parameters) body*

```
LAMBDA(x) x * x * x
```

for the function `cube (x) ≡ x * x * x`

• Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

e.g., `(LAMBDA(x) x * x * x)(2)`

which evaluates to `8`

Higher-order functions

A higher-order function is one that either takes functions as parameters or yields a function as its result, or both.

(1) Function Composition

A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

`f(x) ≡ x + 2` and `g(x) ≡ 3 * x`,

```
h(x) ≡ f(g( x))
     ≡(3 * x)+ 2
```

(2) Apply-to-all:

• Higher-order function that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

Apply-to-all is denoted by α

```
h(x) ≡ x * x
```

`α(h,(2,3,4))` yields `(4,9,16)`

Fundamentals of Functional Programming Languages

•The basic process of computation is fundamentally different in a Functional Programming Language than in an imperative language

– Imperative language: operations are done and the results are stored in variables for later use. Management of variables is a constant concern and source of complexity for imperative programming

– Functional Programming Language, variables are not necessary, as is the case in mathematics.

LISP Data Types and Structures

- The most widely used.
- Two types of data objects in LISP: Atoms and lists

Atoms: either symbols, in the form of identifiers or they are numeric literals

List: parenthesized collections of sublists and/or atoms

e.g., `(A B (C D) E)`

Scheme

- A dialect of LISP, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of LISP

Scheme includes primitive functions for the basic arithmetic operations.

Primitive Functions

- Arithmetic: `+, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX`

function calls were specified in a prefix list form

For example, if + is a function that takes two numeric parameters,

`(+ 5 2)` yields `7`

Special Form Function: `DEFINE`

- **`DEFINE`** – serves two fundamental needs of Scheme programming

1. Bind a symbol to an expression

   e.g., `(DEFINE pi 3.141593)`

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

2. Bind names to expressions, DEFINE takes two lists as parameters.

   The first is the prototype of function call, with the function name followed by the formal parameters, together in a list.

   The second list includes one or more expressions to which the name is to be bound.

The general form of DEFINE:

```
(DEFINE (function_name parameters)
        expression {expression}
)
```

e.g., `(DEFINE (square x)(* x x))`

Example use: `(square 5)`

Example:

Compute the length of the hypotenuse of a right triangle, given the lengths of the two other sides

```
(DEFINE (hypotenuse side1 side2)
  (SQRT(+(square side1)(square side2)))
)
```

Output Functions

- **(DISPLAY expression)**
- **(NEWLINE)**

Numeric Predicate Functions

A predicate function is one that returns a Boolean value

- **#T** is true and **()** is false

Function	Meaning
=	Equal
<>	Not Equal
>	Greater than
<	less than
EVEN?	Is it an even number?
ODD?	Is it an odd number?
ZERO?	Is it zero?
NEGATIVE?	Is it a negative number?

Control Flow: IF

Scheme has two control structure – one for two-way selection and one for multiple selection.

- Two-way Selection

```
(IF predicate then_exp else_exp)
```

e.g.,

```
(IF (<> count 0)
    (/ sum count)
    0)
```

Another Example (factorial function)

```
(DEFINE (factorial n)
   (IF (= n 0)
       1
       (* n (factorial(- n 1)))
))
```

Control Flow: COND

- Multiple Selection

  General form:

```
(COND
   (predicate_1  expr {expr})
   (predicate_2  expr {expr})
      ...
   (predicate_n  expr {expr})
   (ELSE expr {expr})
)
```

The predicates of the parameters are evaluated one at a time, in order from the first until one evaluates to #T.

- Returns the value of the last expr in the first pair whose predicate evaluates to true

In some implementations, ELSE is optional.

Example of COND

```
(DEFINE (compare x y)
  (COND
    ((> x y) (DISPLAY "x is greater than y"))
    ((< x y) (DISPLAY "y is greater than x"))
    (ELSE (DISPLAY "x and y are equal"))
   )
 )
```

List Functions:

The most common use of the Scheme is list processing. List functions deal with lists.

The syntax to define a list is:

'(a b c)

where a, b, and c are constants. We use the apostrophe (') to indicate that what follows in the parentheses is **a list of constant values**, rather than a function or expression.

An empty list can be defined as such:

'()

or simply:

()

List Functions: CONS

• CONS is a primitive list constructor. It builds a list from its two arguments.

CONS takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result

e.g., `(CONS 'A '(B C))` returns `(A B C)`

List Functions: CAR and CDR

- CAR takes a list parameter; returns the first element of that list

  e.g., `(CAR '(A B C))` yields `A`

  `(CAR '((A B) C D))` yields `(A B)`

- CDR takes a list parameter; returns the list after removing its first element

  e.g., `(CDR '(A B C))` yields `(B C)`

  `(CDR '((A B) C D))` yields `(C D)`

Predicate Function: EQ?

•EQ? takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same

e.g., (EQ? 'A 'A) yields #T

(EQ? 'A 'B) yields ()

Predicate Functions: LIST? and NULL?

•LIST? takes one parameter; it returns #T if the parameter is a list; otherwise ()

•NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise ()

Example Scheme Function: member

- member takes an atom and a simple list; returns #T if the atom is in the list; () otherwise

```
DEFINE (member atm lis)
(COND
    ((NULL? lis) '())
    ((EQ? atm (CAR lis)) #T)
    ((ELSE (member atm (CDR lis)))
))
```

//CAR returns the first element of that list

//CDR returns the list after removing its first element

Example Scheme Function: equalsimp

- equalsimp takes two simple lists as parameters; returns #T if the two simple lists are equal; () otherwise

```
(DEFINE (equalsimp lis1 lis2)
(COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((EQ? (CAR lis1) (CAR lis2))
        (equalsimp(CDR lis1)(CDR lis2)))
    (ELSE '())
))
```

Scheme Function: `LET`

A function allows names to be temporarily bound to the values of expressions. These names can then be used in the evaluation of another expression.

- General form:

```
(LET (
    (name_1 expression_1)
    (name_2 expression_2)
    ...
    (name_n expression_n))
   body
)
```

- Evaluate all expressions, then bind the values to the names; evaluate the body

LET Example

```
(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)))
    (DISPLAY (+ minus_b_over_2a root_part_over_2a))
    (NEWLINE)
    (DISPLAY (- minus_b_over_2a root_part_over_2a))
))
```