UNIT-V

Functional Programming Language

Introduction:

- The design of the imperative languages is based directly on the von Neumann architecture.
- The design of the functional languages is based on mathematical functions.

Mathematical Functions:

- A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set.
- A function definition specifies thedomain and range sets, either explicitly or implicitly, along with the mapping.
- The mapping is described by an expression or, in some cases, by a table.
- Mapping expressions is controlled by recursion andconditional expressions.
- Another important characteristic of mathematical functions is that becausethey have no side effects and cannot depend on any external values.
- Mathematical function maps its parameter(s) to a value (or values).

Lambda Calculus

- The Lambda Calculus was developed by Alonzo Church in the 1930s and published in 1941 as ‘The Calculi Of Lambda Conversion’.
- Functions in lambda calculus are very different from those in imperative programming languages (such as Java and C).
- In an imperative programming language the evaluation of a function can have side effects, affecting future evaluations of that function or other functions.
- In lambda calculus a function does not ‘return’ a result based on its parameters — instead the function and its parameters are ‘reduced’ to give an answer, which mathematically is equivalent to the question.
- A function in lambda calculus is written in the form λx.E, where x is the function’s parameter and E is a lambda expression constituting the function body. A lambda expression is either a variable (like the x in the above expression), a function in the form above, or an application E1E2.
- In the expression λx.E, any occurance of x in E is ‘bound’, while any other variable is ‘free’ (unless bound by another lambda expression, like the y in λx.λy.xy). A ‘pure’ lambda expression has no free variables.
- Three things can be done with lambda expressions:
    - α conversion - Alpha conversion renames a bound variable — λx.x can be alpha converted to λy.y.

  - β reduction - Beta reduction allows applications to be reduced — (λx.E1)E2 can be beta reduced to E1 with all occurances of x replaced with E2. If there are name clashes (for example in (λx.λy.xy)y), alpha conversion may be required first.
  - η conversion - Eta conversion allows us to say that f and λx.fx are equivalent.
- Lambda expressions describe nameless functions.
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

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

  which evaluates to 8

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

  Form: h f ° g

  which means h (x) f (g (x))

  For f (x) x + 2 and g (x) 3 * x,

  h f ° g yields (3 * x)+ 2

- A functional form 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

  Form:

  For h (x) x * x

  (h, (2, 3, 4)) yields (4, 9, 16)

Fundamentals of Functional Programming Languages

- The objective of the design of a functional programming language is to mimic mathematical functions to the greatest extent possible.
- The basic process of computation is fundamentally different in a FPL than in an imperative language.
- In an 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.

- A program in an assembly language often must also store the results of partial evaluations of expressions. For example, to evaluate

  (x + y)/(a - b)
- the value of (x + y) is computed first. That value must then be stored while (a - b) is evaluated.
- A purely functional programming language does not use variables or assignment statements.
- Without variables, the execution of a purely functional program has no state in the sense of operational and denotational semantics.
- The execution of a function always produces the same result when given the same parameters. This feature is called referential transparency.
- It also makes testing easier, because each function can be tested separately, without any concern for its context.

The First Functional Programming Language: LISP

LISP Data Types and Structures

•**Data object types:**originally only atoms and lists

•**List form:** parenthesized collections of sublists and/or atoms

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

•Originally, LISP was a typeless language

•LISP lists are stored internally as single-linked lists

LISP Interpretation

•Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.

e.g., If the list (A B C) is interpreted as data it isa simple list of three atoms, A, B, and C

If it is interpreted as a function application,it means that the function named A is applied to the two parameters, B and C

•The first LISP interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation

Programming with Scheme

- The Scheme language, which is a dialect of LISP, was developed at MIT in the mid-1970s (Sussman and Steele, 1975).
- Scheme is a general-purpose computer programming language. It is a high-level language, supporting operations on structured data such as strings, lists, and vectors, as well as operations on more traditional data such as numbers and characters.
- Scheme has been employed to write text editors, optimizing compilers, operating systems, graphics packages, expert systems, numerical applications, financial analysis packages, virtual reality systems, and practically every other type of application imaginable.
- Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts.It repeatedly reads an expression typed by theuser (in the form of a list), interprets the expression, and displays the resultingvalue. This form of interpreter is also used by Ruby and Python.

Primitive Numeric Functions:

- Scheme includes primitive functions for the basic arithmetic operations. Theseare +, −, *, and /. * and + can have zeroor more parameters. If * is given no parameters, it returns 1; if + is given noparameters, it returns 0. + adds all of its parameters together. * multiplies allits 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
(− 24 (* 4 3))	12

- 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. If the parameter is negative, SQRT yields a complex number.

Type Predicates:

- Type predicates type predicate functions:

(boolean? x) ; is x a Boolean?

(char? x) ; is x a character?

(string? x) ; is x a string?

(symbol? x) ; is x a symbol?

(number? x) ; is x a number?

(pair? x) ; is x a (not necessarily proper) pair?

(list? x) ; is x a (proper) list?

- A symbol in Scheme is comparable to what other languages call an identifier.

(symbol? ’x$_%:&=*!) _⇒ #t

The symbol #t represents the Boolean value true. False is represented by #f. Note the use here of quote (’); the symbol begins with x.

Defining Functions:

- A Scheme program is a collection of function definitions.A nameless function actually includes the word LAMBDA, and is calleda lambda expression.

(lambda (x) (* x x)) _⇒function

is a nameless function that returns the square of its given numeric parameter.

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.

- The Scheme special form function DEFINE serves two fundamental needsof Scheme programming: to bind a name to a value and to bind a name to lambda expression.During the evaluation of this expression, x is bound to 7.

form : (DEFINE symbol expression)
For example,
(DEFINE pi 3.14159)
(DEFINE two_pi (* 2 pi))

If these two expressions have been typed to the Scheme interpreter and the pi is typed, the number 3.14159 will be displayed; when two_pi is typed,6.28318 will be displayed.

Numeric Predicate Functions:

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

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 (or #t and #f).

Control Flow:

- Scheme uses three different constructs for control flow: one similar to the selection construct of the imperative languages and two based on the evaluation control used in mathematical functions.
- The Scheme two-way selector function, named IF, has three parameters:a predicate expression, a then expression, and an else expression. A call to IFhas the form

  (IF predicate then_expression else_expression)

  For example,

```
(DEFINE (factorial n)

(IF (<= n 1)

1

(* n (factorial (− n 1)))

))
```

List Functions:

- One of the more common uses of the LISP-based programming languages is list processing.
- Scheme programs are interpreted by the function application function, EVAL. When applied to a primitive function, EVAL first evaluates the parameters of the given function.

  (QUOTE A) returns A
  (QUOTE (A B C)) returns (A B C)
- CAR, which returns the head of a list, cdr ("coulder"), which returns the rest of the list (everything after the head), and CONS, which joins a head to the rest of a list:

  (CAR '(A B C)) returns A

  (CAR '((A B) C D)) returns (A B)

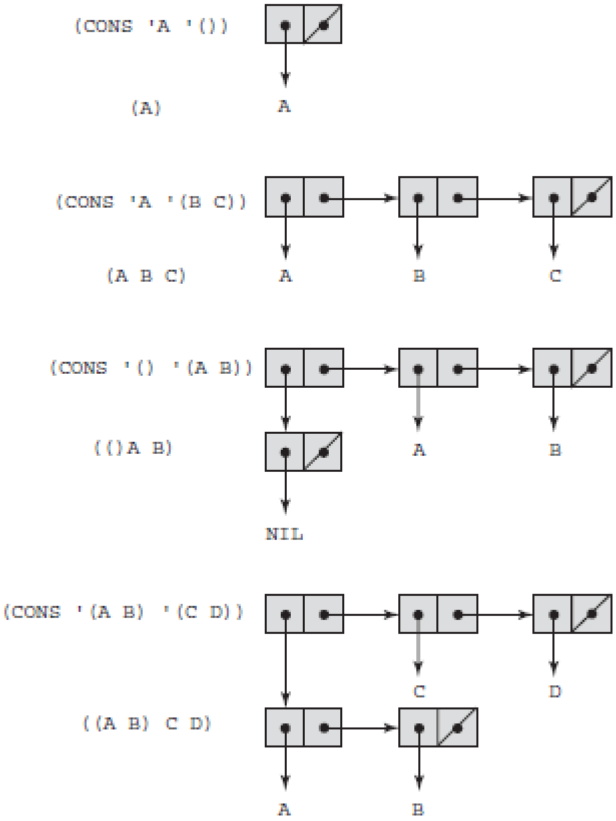
  (CAR'(2 3 4)) _⇒ 2

  (CDR '(2 3 4)) _⇒ (3 4)

  (CONS 2 '(3 4)) _⇒ (2 3 4)

  (CDR'(2)) _⇒ ()

(CONS 2 3) _⇒ (2 . 3) ; an improper list.



The result of several CONS operations

Predicate Functions for Symbolic Atoms and Lists:

Scheme has three fundamental predicate functions, EQ?, NULL?, and LIST?, for symbolic atoms and lists.

- 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. If the two parameters are symbolic atoms, EQ? returns #T if they are the same symbols (because Scheme does not make duplicates of symbols); otherwise #F. Consider the following examples:

  (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

- 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 (initially described in Chapter 5) 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.
- The following example illustrates the use of LET. It computes the roots of a given quadratic equation, assuming the roots are real.8 The mathematical definitions of the real (as opposed to complex) roots of the quadratic equation ax2 + bx + c are as follows:
  root1 = (-b + sqrt(b2 - 4ac))/2a and root2 = (-b - sqrt(b2 - 4ac))/2a.

Functional Composition:

- Functional composition is the only primitive functional form provided by the original LISP. All subsequent LISP dialects, including Scheme, also provide it.
- The function h is the composition function of f and g if h(x) = f(g(x)). For example, consider the following 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. This is an alternative, though less efficient, form of
  CADR. Now consider another call to compose:
  ((compose CDR CAR) '((a b) c d))
  This call would yield (b). This is an alternative to CDAR.
  As yet another example of the use of compose, consider the following:

(DEFINE (third a_list)
((compose CAR (compose CDR CDR)) a_list))
This is an alternative to CADDR.

Programming with ML

- ML (Milner et al., 1990) is a static-scoped functional programming language, like Scheme. One important difference is that ML is a strongly typed language, whereas Scheme is essentially typeless.
- ML has type declarations for function parameters and the return types of functions, although because of its type inferencing they are often not used. The type of every variable and expression can be statically determined.
- ML identifiers do not have fixed types—any identifier can be the name of a value of any type.
- A table called the **evaluation environment** stores the names of all implicitly. and explicitly declared identifiers in a program, along with their types. This is like a run-time symbol table. When an identifier is declared, either implicitly or explicitly, it is placed in the evaluation environment.
- Another important difference between Scheme and ML is that ML uses a syntax that is more closely related to that of an imperative language than that of LISP. For example, arithmetic expressions are written in ML using infix notation.
- Function declarations in ML appear in the general form

  fun function_name(formal parameters) = expression;

  When called, the value of the expression is returned by the function. Actually, the expression can be a list of expressions, separated by semicolons and surrounded by parentheses. The return value in this case is that of the last expression.

  fun circumf(r) = 3.14159 * r * r;

  This specifies a function named circumf that takes a floating-point (**real** in ML) argument and produces a floating-point result. The types are inferred from the type of the literal in the expression. Likewise, in the function

  fun times10(x) = 10 * x;

  the argument and functional value are inferred to be of type **int**.
- Consider the following ML function:

  fun square(x) = x * x;

  In ML, the default numeric type is **int**. So, it is inferred that the type of the parameter and the return value of square is **int**.

  If square were called with a floating-point value, as in

  square(2.75);

  It would cause an error, because ML does not coerce **real** values to **int** type. If we wanted square to accept **real** parameters, it could be rewritten as

  fun square(x) : **real** = x * x;

- ML does not allow overloaded functions, this version could not coexist with the earlier **int** version. The fact that the functional value is typed **real** is sufficient to infer that the parameter is also **real** type. Each of the following definitions is also legal:

  fun square(x : **real**) = x * x;
  fun square(x) = (x : **real**) * x;
  fun square(x) = x * (x : **real**);
- Type inference is also used in the functional languages Miranda, Haskell, and F#.
- The ML selection control flow construct is similar to that of the imperative languages. It has the following general form:

  if expression **then** then_expression **else** else_expression

  The first expression must evaluate to a Boolean value.
- In ML, the particular expression that defines the return value of a function is chosen by pattern matching against the given parameter. For example, without using this pattern matching, a function to compute factorial could be written as follows:

  fun fact(n : **int**): **int** = **if** n <= 1 **then** 1
  else n * fact(n − 1);
- Multiple definitions of a function can be written using parameter pattern matching. The different function definitions that depend on the form of the parameter are separated by an OR symbol (|). For example, using pattern matching, the factorial function could be written as follows:

  fun fact(0) = 1
  | fact(1) = 1
  | fact(n : **int**): **int** = n * fact(n − 1);
- If fact is called with the actual parameter 0, the first definition is used; if the actual parameter is 1, the second definition is used; if an **int** value that is neither 0 nor 1 is sent, the third definition is used.
- ML has a binary operator for composing two functions, o (a lowercase “oh”). For example, to build a function h that first applies function f and then applies function g to the returned value from f, we could use the following:

  val h = g o f;
- ML functions take a single parameter. When a function is defined with more than one parameter, ML considers the parameters to be a tuple, even though the parentheses that normally delimit a tuple value are optional. The commas that separate the parameters (tuple elements) are required.
- ML functions that take more than one parameter can be defined in curried form by leaving out the commas between the parameters (and the delimiting parentheses).11 For example, we could have the following: **fun** add a b = a + b; Although this appears to define a function with two parameters, it actually defines one with just one parameter. The add function takes an integer parameter (a) and returns a function that also takes an integer parameter (b).

- A call to this function also excludes the commas between the parameters, as in the following:

```
add 3 5;
```

  This call to add returns 8, as expected.
- Curried functions also can be written in Scheme, Haskell, and F#. Consider the following Scheme function:

```
(DEFINE (add x y) (+ x y))
```

  A curried version of this would be as follows:

```
(DEFINE (add y) (LAMBDA (x) (+ y x)))
```

  This can be called as follows:

```
((add 3) 4)
```

- ML has enumerated types, arrays, and tuples. ML also has exception handling and a module facility for implementing abstract data types.

Important Questions

1.) Discuss the fundamental concepts of Lambda Calculus? (2017 SET-1 8M)
2.) Explain about LISP functional programming language? (2017 SET-1 8M)
3.) How ML is different from other functional programming language? (2017 SET-2 8M)
4.) Why were imperative features added to most dialects of LISP? (2017 SET-1 8M)
5.) Write a short note on ML functions? (2017 SET-1,2 3M)
6.) Explain about Scheme functional programming language? (2017 SET-3 8M)
7.) Give comparison of Functional and Imperative Languages? (2017 SET-4 8M)
8.) Explain in detail about Functional Programming Language?
9.) Explain how functions are defined in Scheme and ML? (2016 SET-4 8M)
10.)Explain about list and primitive functions in Scheme? (2016 SET-3 16M)
11.) Explain about Predicate functions in Scheme? (2016 SET-2 8M)
12.) Explain about data objects in LISP? (2016 SET-1 12M)