Functional programming Languages Unit-5

- The functional programming paradigm, which is based on mathemat. functions, is the design basis of the most important non-imperative styles of languages.
- Purely functional programming languages are better than imperative languages because they result in programs that are more readable, ma reliable and more likely to be correct.
- The main disadvantage of imperative language program is that the state of the program, which changes throughout the execution process is maintained by the program
  - The state is represented by program variables
  - The author and all readers of the program must understand the uses of its variables and state changes to understand the program.

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 the domain set, the range set and the mapping between the sets.
  - The mapping is described by an expression.
- The domain set may be the cross product of several sets (meaning a function can be having more than one parameter).
- An important characteristic of mathematical functions is that ~~because~~ they have no side ~~effect~~ effects and cannot depend on any external value.
- A mathematical function maps its parameters to a value, rather than specifying a sequence of operations on variables.

Simple functions

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

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

    ∴

    'is defined as'

- Lambda notation, provides a method for defining nameless functions
- A lambda expression specifies the parameters and the mapping of a function

  λ(x) x * x * x

- A formal computational model using lambda expressions is called lambda calculus. In this model, the function definition, function applications and execution are implemented using lambda expressions.
- Application of a lambda expression is done by the following example

  (λ(x) x * x * x (2))

  - results in 8.

Function forms | Higher order functions

- A higher order function is one that either takes one or more functions as parameters or results a function.

1) Function composition

- function composition take two functional parameters and yields a function whose value is the first parameter function applied to the result of second parameter function.

Ex: $f(x): x+2 \quad g(x): 3*x$

$h(x): f \circ g = f(g(x)) = (3*x)+2$

2) Apply-to-all

- Apply to all takes a unary function as a parameter.
- If applied to a list of arguments, apply-to-all applies its function parameter to each of the values in the list argument.

Let $h(x): x*x$

then

$\alpha(h,(2,3,4))$ yields $(4,9,16)$.

Functional vs Imperative languages

1) Functional languages have very simple syntax structures compared to imperative langu
2) FL's have simple semantics compared to IL's
3) The size of FL's programs is very small compared IL's.
4) The readability of a functional language program is better compared to IL's as the FL programs hide the many implementation details.
   - In an imperative language, an expression is evaluated and the result is stored in a memory location, which is referenced as a variable in a progra this memory cell is the state of a program. This is a low-level programming methodology
   - A purely functional programming language does not use variables or assignment statements, so the programmer is concerned about the variables or memory cells.

FL programs implement recursion but not iteration

The execution of a function always produces the same result when given the same parameter. This feature is called referential transparency, which exist in FPL not in IP

FL programs are easier to test for errors

Scanned by CamScanner

Scheme

- Scheme is relatively simple, it is popular in colleges and universities for educational purpouse.

- Scheme is
  1) Small in size
  2) Typeless
  3) Simple syntax and semantics
  4) uses static scoping.
  5) Functions are treated as first-class entities.

- As first-class entities, scheme functions can be the values of expressions, elements of lists, passed as parameters and returned from functions.

Scheme interpreter

- A scheme interpreter in interactive mode is an ~~and~~ infinite read-evaluate print loop (REPL).
- It repeatedly reads an expression typed by the user, interprets the expression and displays the resulting value.
- 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.

Expression	Value
42	42
(* 3 7)	21

(- 5 6) -1

(+ 15 7 2) 6

(- 24 (* 4 3)) 17

- * and + can have zero or more parameters
- If * has no parameters it returns 1 and if + is given no parameters it return '0'.
- / and - have two or more parameters
- There are large number of other numeric functions in Scheme, among them MODULO, ROUND, MAX, MIN, LOG, SIN and SQRT

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

(LAMBDA (x) (* x x)) - returns the square a number

(LAMBDA (x) (* x x)) 7) - returns 49

- Lambda expressions can have any number of parameters

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

Define function

- The Scheme has a special higher order function DEFINE
- DEFINE has two fundamental needs for programmers
  1) To bind a name to a value
  2) To bind a name to a lambda expression.

- DEFINE is called a special form because it is interpreted in a different way than the normal primitives like the arithmetic functions.

- Examples

1) - To bind a name to a value

(DEFINE Symbol expression)

- (DEFINE pi 3.142)
(DEFINE two_pi (* 2 pi))

2) To bind a name to a lambda expression

(DEFINE (function_name parameters)
(expression)
)

- (DEFINE (square number)
(number * number)
)

→ After interpreter evaluates this function, it can be used as

(square 5)

25

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

Output Functions

- Scheme includes a few simple output functions.
- Scheme includes a formatted output function, PRINTF, which is similar to printf function in C.
- A pure functional programming model should not contain explicit input or output functions, because input operations change the program state and output operations have side effects.

Numeric predicate functions

- A predication function is one that returns a Boolean value (#t or #f)
- Scheme includes a collection of predicate functions for numeric data.

Function	Meaning
=	equal
<>	not equal
>	Greater than
<	less than
>=	Greater than equal to
<=	less than equal to
Even?	Is it an even number?
ODD?	Is it odd number?
zero?	Is it zero?

- In Scheme the two Boolean values are #t and #f (or #t or #f).

Control Flow

- Scheme uses three different constructs for control flow.

1) The Scheme two-way selector function It has three parameters.

(If predicate then_expression else_expression)

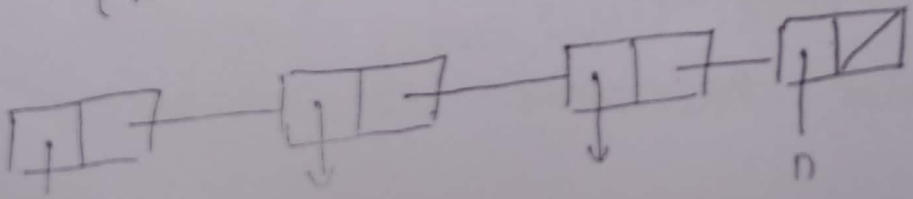
Example

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

List Functions

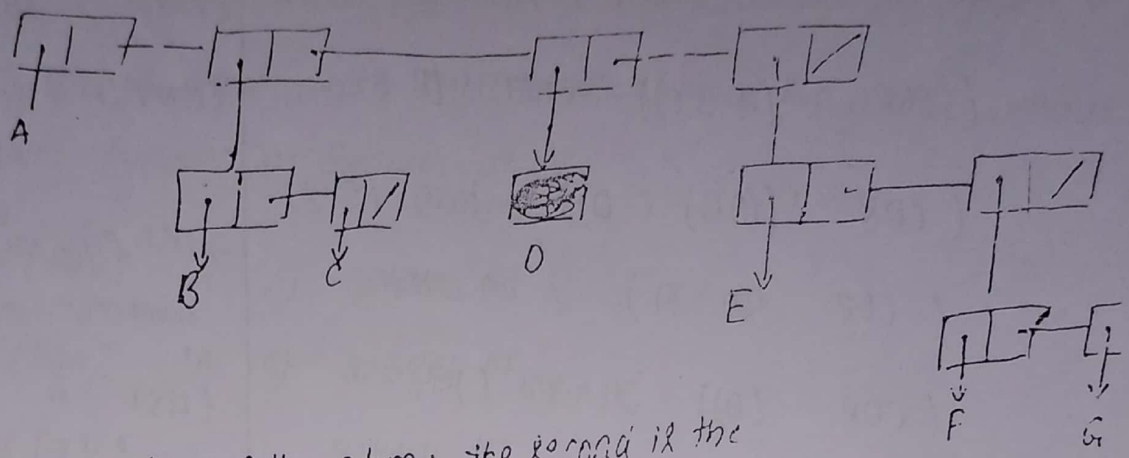
- Scheme functions deals with list just like LISP-programming
- There are two categories of data objects in LISP:
    1) Atoms
    2) Lists.
- List elements are pairs, where the first part is the data of the element, which is a pointer to either an atom or a nested list.
  - The second part of a pair can be a pointer to an atom, a pointer to another element or an empty list.
- Elements are linked together in lists with the second parts.
- Atoms are either symbols or numberic literals.
- Lists are specified in LISP by delimiting their elements with parantheses.
- Simple lists only contain atoms:

(A B C D)



- Nested list structures are also specified by parentheses.

(A (B C) D (E (F G)))



- The first element A is the atom; the second is the sublist (B C).

→ Scheme defines two functions

- CAR - contents of the address part of a register
- CDR - contents of the decrement part of a register.

- To avoid treating the list as parameters, first the list is given as a member parameter to the primite function QUOTE.

~~(QUOTE (A B))~~

(QUOTE A) returns A

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

- Instead of writing (QUOTE (A B C)) we write '(A B C)

→ The CAR and CDR operations are as follows:

(CAR '(A B C)) return A

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

(CAR 'A) is an error because A is not a list

(CAR '(A)) return A

(CAR '()) is an error.

(CDR '(A B C)) return (B C)

(CDR '((A B) C D)) return (C D)

(CDR 'A) is an error.

(CDR '(A)) return ()

(CDR '()) is an error.

→ CONS creates a new list from existing list.

(CONS '(B C)) return (A B C)

(CONS '(A B) '(C D)) return ((A B) C D)

→ LIST function creates a list from various number of elements

(LIST 'A 'B '(C)) return (A B (C)).

Predicate Functions for Symbolic Atoms and Lists

- Scheme has three fundamental predicate functions

1) EQ?
2) NULL?
3) LIST?

- The EQ? function takes two expressions as parameters. It returns #T if both parameters have the same pointer value ie they are the same atom or list; otherwise it returns #F.

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 last two examples show that EQ? function is not consistent while dealing with numeric atoms.

- EQ? works for symbolic atoms, but not necessarily correct for no atoms.

- The '=' predicate works for numeric atoms, but not for symbolic at

- EQV? is a predicate, which tests the equality of two atoms without considering numeric or symbolic atoms.

(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.

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

(LIST? 'X) returns #F

(LIST? '()) returns #T

The NULL? predicate function returns #T if the single parameter is an empty list.

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

(NULL? '()) returns #T

(NULL? 'A) returns #F

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

- The last call returns #F because the parameter contains a single element, ie a empty list.

Example Scheme functions:

- There are various list processing problems, which are solved by functions in Scheme.

1) Membership of a given atom in a given list that donot include sublists. Such a list is called a simple list.

- If the function is named member, it could be used as follows:

(member 'B '(A B C)) returns #T

(member 'B '(A C D E)) returns #F

- This function is defined as follows.

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

- The individual elements are specified with CAR and the process is continued using recursion on the CDR of the list.

2) The problem of determining whether two given lists are equal or not (simp

```
(DEFINE (equalsmp list1 list2)
  (COND
    ((NULL? list1) (NULL? list2))
```

```
((EQ? (CAR list1) (CAR list2))
        (equalSimp (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

3) Another list operation is that which constructs a new list that contains all of the elements of two given list arguments. This is implemented as Scheme function append.

(append '(A B) '(C D E)) returns (A B C D E)

(append '((AB) C) '(D (E F))) returns ((A B) C D (E F))

- The definition of append is as follows:

```
(DEFINE (append list1 list2)
    (COND
        ((NULL? list1) list2)
        (ELSE (CONS (CAR list1) (append (CDR list1) list2)))
    )
)
```

LET

LET is a function that creates a local scope in which names are temporarily bound to the values of expressions.

Here names can then be used to evaluate another expression.

Ex The real roots of a quadratic equation are

$$root1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad root2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

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

Tail Recursion in Scheme

A function is tail recursive if its last operation in the functional is a recursive call.

- Member function with tail recursion

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

Factorial of a number using tail recursion can be done using a helper function

Recursion

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

- Here the last operation is multiplication.

Tail recursion

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

Higher order Functions/ Functional Forms

1) Functional composition

- The function h is the composition function of f and g if $h(x) = f(g(x))$

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

→ $h = f \circ g = f(g(x))$

- The functional composition function compose can be written as follows:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
```

2) Apply-to-All function

```
(DEFINE (map fun a_list)
    (COND
        ((NULL? a_list) '())
        (ELSE (CONS (fun (CAR a_list))
                    (map fun (CDR a_list))))
    )
)
```

Functions that Build Code

- Scheme interpreter uses a function named EVAL to evaluate the expression.
- This EVAL function can also be called directly by Scheme programs.
- Example
  - Generally (+ 3 7 10 2) return 22.
  - But the problem is that in a program with list of numeric atoms. Adding the number atoms in the list is not directly possible on a list using '+'
  - Such an operation can be implemented in the following manner.

```
(DEFINE (adder a_list)
    (COND
        ((NUL? a_list) 0)
```

- An alternative solution is to write a function that builds a list and use EVAL function.

```
-(DEFINE (adder a_list)
   (COND
      ( (NULL? a_list) 0)
      (ELSE (EVAL (CONS '+ a_list)))
   ))
```

ML

- ML is a static-scope functional programming language like scheme. proposed by Milner, in 1990.
- one important difference of Schema and ML is that ML is a strongly typed language, where as scheme is essentially typeless language.
- In ML, the data type of every variable and expression can be statistically determined.
- ML, like other functional programming languages do not have variables, but do have identifiers, which have the appearance of names of variables in imperative languages.
- These identifiers are thought of as names for values. once set they cannot be changed. These are like final declarations in java.
- A table called the evaluation environment stores the names of all declared identifiers in a program, along with their types.

- Another important difference between Scheme and ML is that ML uses a syntax that is more closely related to that of an imperative language.

Ex

```
fun function_name (formal parameters) = expression;
```

Ex.

```
fun square(x) = x*x;
fun circum(r) = 3.14159 * r * r;
```

- ML is a typed language and the default numeric type is int.

-Considers the ML function

```
fun square(x) = x*x;
```

- In this ML determines the type of both the parameters, which by default is int.

- If the square were called with a floating-point value

```
square(2.75)
```

-It would cause an error.

- If we wanted square to accept real parameters it could be rewritten as:

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

→ The following definitions is also legal

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

Control flow

- The ML selection control flow construct is similar to that of imperative languages.

  - if expression then then_expression else else_expression

- The first expression must evaluate to a boolean value.

- 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 definition of a function can be written using parameter pattern matching. It is done by using OR (|) symbol.

- The factorial could be re-written as follows:

```
fun fact(0) = 1
| fact(1) = 1
| fact(n:int) : int = n * fact(n-1);
```

Lists

- ML supports lists and list operations.

ML's version of Scheme's CAR, CDR, CONS are hd, tl and ::.

But these are less frequently used compared to Scheme's CAR, CDR, CONS.

Ex:-

ML program to calculate the length of a list is as follows:

```
fun length([]) = 0
  | length(h::t) = 1 + length(t);
```

- The first case in this function handles the situation of the function being called with an empty list as the first parameter.
- The second case of the function breaks the list into its head and tail.
- Another example of append function is as follows:

```
fun append([], list2) = list2
  | append(h::t, list2) = h :: append(t, list2);
```

Identifiers

- In ML, values are bound to names, with value declaration statements.

```
val new_name = expression;
```

Ex:

```
val distance = time * speed;
```

- The val statement binds a name to a value, but the name cannot be later rebound to a new value.

Higher-order functions

ML includes several higher-order functions that are commonly used in functional programming.

1) filter

- filter takes a list as a parameter and returns only those values for which the predicate function is true.

- The predicate function is defined exactly like a function, except with "fn" reserved word instead of fun.

- filter (fn(x) ⇒ x<100, [25, 1, 50, 711, 100, 150, 27, 161, 3]);

returns [25, 1, 50, 27, 3].

2) map, Apply all

- The map function takes a single parameter, which is a function
- The result function takes a list as a parameter. It applies its function to each element of the list and returns a list of the results.

- fun cube x = x * x * x;

val cubelist = map cube;

val newlist = cubelist [1, 3, 5];

returns [1, 27, 125].

- The same could be achieved by the following:

val newList = map (fn x ⇒ x * x * x, [1, 3, 5]);

3) Composition

- ML has a binary operation for composing two function, a lowercase 'o'.

- val h = f o g;

- In this it first applies function f and then applies function g.

4) Currying

- Currying replaces a function with more than one parameter, with a function with one parameter that returns a function that takes the other parameters of the initial function.

Ex: fun add a b = a + b;

- Although the above function appears to define a function with two parameters, it actually defines one with just one parameter.
- Strictly speaking ML functions takes a single parameter.
- The "add" function takes an integer parameter (a) and returns a function that also takes an integer parameter (b).
- The call will be as follows.

add 3 5; returns 8

→ Curried functions can be used to construct other useful functions, such as the following.

- fun adds x = add 5 x;
- val num = adds 10;
- The value of num is 15

- ML also have enumerated types, array, tuples, exception handling and also facility to implement abstract data types.

— x —