PPL - UNIT V

Functional Programming Languages

- 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

eg: LISP, scheme, common Lisp, Haskell, Miranda

Mathematical Functions

Def: A mathematical function is a mapping of members of one set, called the *domain set*, to another set, called the *range set*

Introduction to lambda calculus

- The Lambda Calculus was invented by Alonzo Church [1932] as a mathematical formalism for expressing computation by functions.
- The word lambda is the name of the Greek symbol “λ” and is borrowed from Lambda calculus
- Any computable function can be written as lambda expression
- Lambda notation is used to specify functions and function definitions, function applications, and data all have the same form
- Lambda expressions describe nameless functions

A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

l(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. (l(x) x * x * x)(3)

which evaluates to 27

Functional Forms

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

There are two types of mathematical Functional Forms

1. Function Composition
2. Apply-to-all

1. *Function Composition*

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

Function Composition is written as an exp using ° as an operator, as in $h \equiv f \circ g$

which means $h(x) \equiv f(g(x))$

$\equiv$ (defined as)

For example if $f(x) \equiv x + 2$
$g(x) \equiv 3 * x$, then
$h(x) \equiv f(g(x))$ or $h(x) \equiv (3 * x) + 2$

2. *Apply-to-all*

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

Apply-to-all is denoted by α

Let $h(x) \equiv x * x$

then

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

Fundamentals of FunctionalProgramming Languages

In FunctionalProgramming everything is a function

- The objective of the design of a FPL 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 a memory location, which is represented as a variable in program
  - Management of variables is a constant concern and source of complexity for imperative programming
  - In an FPL, does not use variables or assignment statements, as is the case in mathematics
  - In an FPL, variables are not necessary, as is the case in mathematics
  - In an FPL, the evaluation of a function always produces the same result given the same parameters. This is called *referential transparency*

Referential transparency means that *“equals can be replaced by equals”*.
In a pure functional language, all functions are referentially transparent, and therefore *always yield the same result* no matter how often they are called.

LISP

- LISP is the first functional Programming Language
- **LISP = LIS**t **P**rocessing
  - Invented in 1959 by John McCarthy
- **LISP has only two kinds of data objects: 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
- Lists are specified by delimiting their elements with parentheses. Simple lists, in which elements are restricted to atoms, have the form (A B C D)
- Nested list structures are also specified by parentheses.

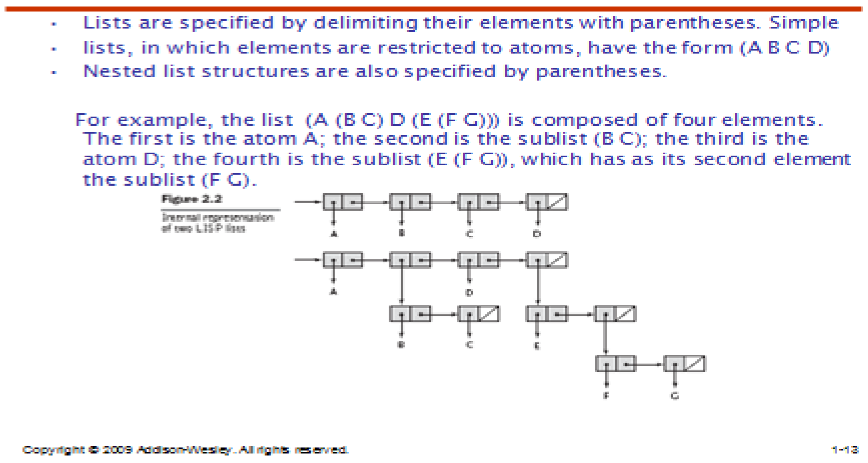
<u>LISP Interpreter or Basics of LISP</u>

If we want to write an interpreter for LISP then LISP functions must be expressible as lamda expressions or S-expressions and they are lists

e.g., If the list (A B C) is interpreted as data it is a 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

x + y :: = (PLUS X Y)

2x + 3y :: = (PLUS (TIMES 2 x) (TIMES 3 y))



- Lists are specified by delimiting their elements with parentheses. Simple
- lists, in which elements are restricted to atoms, have the form (A B C D)
- Nested list structures are also specified by parentheses.

For example, the list (A (B C) D (E (F G))) is composed 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 2.2
Internal representation of two LISP lists

Copyright © 2009 Addison-Wesley. All rights reserved. 1-13

LISP Interpreter make use of LISP basic functions *car, cdr, cons, append, member.*

LISP basic Functions

The following table provides some commonly used list manipulating functions.

Function	**Description**
car	It takes a list as argument, and returns its first element.
cdr	It takes a list as argument, and returns a list without the first element.
cons	It takes two arguments, an element and a list and returns a list with the element inserted at the first place.
list	It takes any number of arguments and returns a list with the arguments as member elements of the list.
append	It merges two or more list into one.
last	It takes a list and returns a list containing the last element.
member	It takes two arguments of which the second must be a list, if the first argument is a member of the second argument, and then it returns the remainder of the list beginning with the first argument.
reverse	It takes a list and returns a list with the top elements in reverse order.

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

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


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

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

Scheme language

scheme is a descendent of LISP
It is functional programming language
Uses only static scoping
- Functions are first-class entities
  - They can be the values of expressions and elements of lists
  - They can be assigned to variables and passed as parameters

Elements of scheme

- All programs and data in scheme are expressions and expressions are of two varieties: atoms and lists

- Atoms include numbers, strings, names etc

Syntax of scheme

```
 exp  -> atom | list
 atom -> number | string | identifier |
              character | boolean
 list  -> '(' exp-sequence')'
exp-sequence -> exp exp-sequence   | exp
```

Examples of scheme expressions

42- a number

“hello” - a string

#T   - boolean value “true”

#\a   - character a

(2 3 4) - list of numbers

a   - identifier

hello - another identifier

(+ 2 3) – list

(*(+ 2 3) (/ 6 2)) - list

Evaluation

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function
- 3+4*5   in   C
- (+ 3 (* 4 5)) in scheme

Primitive Functions

1. Arithmetic: +, -, *, /, ABS, SQRT e.g., (+ 5 2) yields7

2. QUOTE -takes one parameter; returns the parameter without evaluation

QUOTE is used to avoid parameter evaluation when it is not appropriate.
QUOTE can be abbreviated with the apostrophe prefix operator
e.g., '(A B) is equivalent to (QUOTE (A B))

List Functions

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)

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 - takes any number of parameters; returns a list with the parameters as elements
LIST('A'B'C) returns (A B C)

Predicate Functions*:* (#T and () are true and false)

1. 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 '(A B)) yields ()
   – Note that if EQ? is called with list parameters, the result is not reliable
   – Also EQ? does not work for numeric atoms

2. LIST? takes one parameter; it returns #T if the parameter is an list; otherwise ( )

   e.g

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

3. NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise ()
   Note that NULL? returns #T if the parameter is ()

- (NULL? '(A B)) returns #F
- (NULL? '()) returns #T
- (NULL? 'A) returns #F
- (NULL? '(())) returns #F

3. **Numeric Predicate Functions**
   A Predicate Function is one that returns a boolean value either true or false

=, < >, >, <, >=, <=, EVEN?, ODD?, ZERO?

Input/ Output Functions:

(DISPLAY expression) - O/P FUNCTION
eg: (DISPLAY "HELLO")

(NEWLINE)

(READ) - I/P FUNCTION

Defining functions

Lambda Expressions are used to specify functions in scheme

- A scheme program is a collection of functions. scheme functions are based on lambda notation
- A lambda expression has the following form
- (lambda param-list body)

(lambda (x) (* x x))

here x is a bound variable

((lambda(x)(* x x)) 5)

which evaluates to 25

DEFINE is a Function for namingFunctions

DEFINE - Two forms:

1. To bind a symbol to an expression e.g.,

(DEFINE pi 3.141593)
(DEFINE two_pi (* 2 pi))

2. To bind names to lambda expressions

(DEFINE square (lambda (x) (* x x))

In this case lambda expression is abbreviated by removing the word lambda so it is completely equal to
(DEFINE (square x) (* x x))

- *Evaluation process (for normal functions):*
  1. Parameters are evaluated, in no particular order

2. The values of the parameters are substituted into the function body
3. The function body is evaluated
4. The value of the last expression in the body is the value of the function

- *Control Flow*

- *1. **Selection**- the special form,* IF

```
(IF predicate then_exp else_exp) e.g.,

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

- 2. *Multiple Selection* - the special form, COND
  - General form:

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

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

Eg:

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

Example Scheme Function: LET

- 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**

Eg:

```
(LET (( a 3)
      ( b 4)
    (square (lambda (x) (* x x)))
    (plus +))
  (sqrt(plus(square a) (square b)))) => 5
```

Example Scheme Functions

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

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

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

2. **append** - takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

Eg: (append '(A B) '(C D R)) returns (A B C D R)

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

Tail Recursion in Scheme

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration
- Example of rewriting a function to make it tail recursive, using helper a function

A function call is said to be tail recursive if there is nothing to do after the function returns except return its value. Since the current recursive instance is done executing at that point, saving its stack frame is a waste. Specifically, creating a new stack frame on top of the current, finished, frame is a waste. A compiler is said to implement TailRecursion if it recognizes this case and replaces the caller in place with the callee, so that instead of nesting the stack deeper, the current stack frame is reused.

Original: (DEFINE (factorial n)

```
        (IF (= n 0)
          1
          (* n (factorial (- n 1))))
      (fact 5))          =>   120
```

Tail recursive:
```
(DEFINE (factorial1 n acc)
        (IF (= n 0)
          acc
          factorial1((- n 1) (* n acc)))
      ))
(DEFINE (factorial n)
      (factorial1 n 1))
```

tail recursion in C

```
factorial1(n, accumulator)
{
if (n == 0) return accumulator;
return factorial1(n - 1, n * accumulator);
}

factorial(n)
{
return factorial1(n, 1);
}
```

Functional Forms provided by scheme are:

- Function Composition
  - The previous examples have used it
  - (CDR (CDR '(A B C))) returns (C)
  - (CAR(CAR '((A B)B C))) returns A
- Apply to All - one form in Scheme is map
  - Applies the given function to all elements of the given list;
    (map (LAMBDA (num) (* num num num)) '(3 4 2 6))

  This call returns (27 64 8 216).
  Note that in this example, the first parameter to map is a LAMBDA expression

Currying in scheme

A Curried function [named after the logician H.B. Curry] *takes its arguments one at a time*, allowing it to be treated as a higher-order function.

- A common operation , named after logician Haskell Curry , is to repalce a multiargument function with a function that takes a single argument and returns a function that expects the remaining arguments
  (DEFINE (add x y) (+ x y))
  A curried version of above would be as follows:
  (DEFINE (add y) (LAMBDA (x) (+ y x)))
  This can be called as ((add 3) 4)

Example scheme programs

Factorial program in scheme

```
(DEFINE (factorial n)
   (IF (= n 0)
     1
   (* n (factorial (- n 1))))
  (factorial 5))           =>   120
```

gcd program in scheme

```
(DEFINE (gcd a b)
        (cond (( = a b) a )
                  (( >a b) (gcd ( - a b) b))
                   ((else (gcd ( -b a) a)))
    )   )
```

Eg: (gcd25 10)
5

GCD program in C

```
int gcd( int a, int b)
  {
    if(a==b) return a
    else if(a>b) return gcd(a-b,b)
   else return gcd(a,b-a)
}
```

ML (Meta language)

- A static-scoped functional language with syntax that is closer to Pascal than to LISP
- Uses type declarations, but also does type inferencing to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types

- Includes lists and list operations
- The val statement binds a name to a value (similar to DEFINE in Scheme)
- ML does not allow overloaded functions

- **SYNTAX of Function in ML:**

```
fun function_name (formal_parameters) = function_body_expression;

e.g., fun cube (x : int) = x * x * x;

      fun areac( r ) = 3.14*r*r;
```

In the above function type of parameter is absent. In such cases type inference is used i.e types are obtained from the type of constant(3.14) in the expreesion. so, type of r is real

Consider the following ML function

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

here type of x is not specified,.In ML the default numeric type is int.

If square is called with
square(2.75);

It would cause an error. If we wanted square to accept real parameters, it could be written as

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

Factorial function in ML

```
fun fact(n: int): int = if n=0 then 1
                  else n * fact(n – 1);
val fact = fn: int -> int
```

ML responds that the value of fact is fn, with type int -> int, which means that fact is a function from integers to integers

Once a function has been declared it can be called as follows:

```
>fact 5;
val it = 120 : int
```

- **ML selection**

```
if expression then then_expression
  else else_expression
```

where the first expression must evaluate to a Boolean value

- Pattern matching is used to allow a function to operate on different parameter forms

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

- **Lists**

  Literal lists are specified in brackets

  [3, 5, 7]

  [ ] is the empty list

  CONS is the binary infix operator, ::

  4 :: [3, 5, 7], which evaluates to [4, 3, 5, 7]

  CAR is the unary operator hd

  CDR is the unary operator tl

```
>hd [1,2,3];
val it = 1: int
> tl [1,2,3];
val it = [2,3] : int list
```

- The val statement binds a name to a value (similar to DEFINE in Scheme)

```
> val pi = 3.14159
val pi = 3.14159 : real
```

- val is nothing like an assignment statement in an imperative language

HASKEL

Similar to ML (syntax, static scoped, strongly typed, type inferencing, pattern matching)

Different from ML (and most other functional languages) in that it is *purely* functional (e.g., no variables, no assignment statements, and no side effects of any kind)

- **Some key features**:
  - Uses lazy evaluation (evaluate no subexpression until the value is needed)
  - **Haskel** has **"list comprehensions,"** and **infinite lists**

List Comprehension of Haskel

- **List of the squares of the first 20 positive integers: [n * n | n ← [1..20]]**
- **All of the factors of its given parameter:**

  factors n = [i | i ← [1..n`div` 2],

n`mod` i == 0]

Lazy evaluation

- Lazy evaluation means that expressions are evaluated Only if their values are needed
- A language is *strict* if it requires all actual parameters to be fully evaluated
- A language is *nonstrict* if it does not have the strict requirement
- Nonstrict languages are more efficient and allow some interesting capabilities – *infinite lists*
- Nonstrict languages can use an evaluation form called Lazy evaluation

HASKEL PROGRAMS for factrorial and fibonacci

```
fact(0) = 1
fact(n) = n * fact (n - 1)
```

fib 0= 0

```
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

Applications of Functional Languages:

- LISP is used for artificial intelligence
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
- Scheme is used to teach introductory programming at a significant number of universities

Comparing Functional and Imperative Languages

- ***Imperative Languages:***
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed

- ***Functional Languages:***
  - Simple semantics
  - Simple syntax
  - Inefficient execution

- Programs can automatically be made concurrent