UNIT-5
Functional Programming Languages & Scripting Language

5.1 Functional Programming Language Introduction – CO5

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

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

  λ(x) x * x * x

  for the function cube (x) = x * x * x

Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., (λ(x) x * x * x)(2)

  which evaluates to 8

Functional Forms

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

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

  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

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

  Form:α
  For h (x) ≡ x * x
  α(h, (2, 3, 4)) yields (4, 9, 16)

5.2 Fundamentals of Functional Programming Languages- CO5

- 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 variables for later use
- Management of variables is a constant concern and source of complexity for imperative programming

- In an FPL, variables are not necessary, as is the case in mathematics

Referential Transparency

- In an FPL, the evaluation of a function always produces the same result given the same parameters

The First Functional Programming Language : LISP – CO5

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

5.3 ML – CO5

- 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

ML Specifics

- Function declaration form:
  fun name (parameters) = body;
  e.g., fun cube (x : int) = x * x * x;
  - The type could be attached to return value, as in
    *fun cube (x) : int = x * x * x;*
  - With no type specified, it would default to
    int (the default for numeric values)
  - User-defined overloaded functions are not allowed, so if we wanted a cube function for real parameters, it would need to have a different name
  - There are no type coercions in ML
- 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(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

```
fun length([]) = 0
| length(h :: t) = 1 + length(t);
fun append([], lis2) = lis2
| append(h :: t, lis2) = h :: append(t, lis2);
```

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

  val distance = time * speed;

  – As is the case with DEFINE, val is nothing like an assignment statement in an imperative language

5.4 Haskell – CO5

- 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)

  Syntax differences from ML

```
fact 0 = 1
fact n = n * fact (n – 1)
    fib 0 = 1
    fib 1 = 1
        fib (n + 2) = fib (n + 1) + fib n
```

Function Definitions with Different Parameter Ranges

```
fact n
| n == 0 = 1
| n > 0 = n * fact(n – 1)

sub n
| n < 10 = 0
| n > 100 = 2
| otherwise = 1
square x = x * x
```

- Works for any numeric type of x

Lists

- List notation: Put elements in brackets

  e.g., directions = ["north", "south", "east", "west"]

- Length: #

  e.g., #directions is 4

- Arithmetic series with the .. operator

  e.g., [2, 4..10] is [2, 4, 6, 8, 10]

- Catenation is with ++

  e.g., [1, 3] ++ [5, 7] results in [1, 3, 5, 7]

- CONS, CAR, CDR via the colon operator (as in Prolog)
  e.g., 1:[3, 5, 7] results in [1, 3, 5, 7]

Factorial Revisited

```
product [] = 1
product (a:x) = a * product x
fact n = product [1..n]
```

List Comprehension

- Set notation
- 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]
```

Quicksort

```
sort [] = []
sort (a:x) =
sort [b | b ← x; b <= a] ++
[a] ++
sort [b | b ← x; b > a]
```

Lazy Evaluation

- 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*
- Lazy evaluation - Only compute those values that are necessary
- Positive numbers

  positives = [0..]
- Determining if 16 is a square number

```
member [] b = False
member(a:x) b=(a == b)||member x b
squares = [n * n | n ← [0..]]
member squares 16
```

Member Revisited

- The member function could be written as:

```
member [] b = False
member(a:x) b=(a == b)||member x b
```

- However, this would only work if the parameter to squares was a perfect square; if not, it will keep generating them forever. The following version will always work:

```
member2 (m:x) n
| m < n = member2 x n
| m == n = True
| otherwise = False
```

5.5 Applications of Functional Languages – CO5

- APL is used for throw-away programs
- 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 some 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

Summary of Functional Programming Languages

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution instead of imperative features such as variables and assignments
- LISP began as a purely functional language and later included imperative features
- Scheme is a relatively simple dialect of LISP that uses static scoping exclusively
- COMMON LISP is a large LISP-based language
- ML is a static-scoped and strongly typed functional language which includes type inference, exception handling, and a variety of data structures and abstract data types
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- Purely functional languages have advantages over imperative alternatives, but their lower efficiency on existing machine architectures has prevented them from enjoying widespread use

Pragmatics

A software system often consists of a number of subsystems controlled or connected by a script. ***Scripting*** is a paradigm characterized by:

- Use of scripts to glue subsystems together.
- Rapid development and evolution of scripts.
- Modest efficiency requirements.
- Very high-level functionality in application-specific areas.

Key Concepts

The following concepts are characteristic of scripting languages:

- Very high-level string processing.
- Very high-level graphical user interface support.
- Dynamic typing.

Case Study: PYTHON

- PYTHON was designed in the early 1990s by Guido van Rossum.
- It has been used to help implement the successful Web search engine GOOGLE, and in a variety of other application areas ranging from science fiction (visual effects for the *Star Wars* series) to real science (computer-aided design in NASA).

Values and Types

- PYTHON has a limited repertoire of primitive types: integer, real, and complex numbers.
- It has no specific character type; single-character strings are used instead. Its boolean values (named False and True) are just small integers.
- PYTHON has a rich repertoire of composite types: tuples, strings, lists, dictionaries and objects. A PYTHON list is a heterogeneous sequence of values.
- A *dictionary* (sometimes called an associative array) is a heterogeneous mapping from keys to values, where the keys are distinct immutable values.
- The following code illustrates tuple construction:

```
date = 1998, "Nov", 19
```

  Now date[0] yields 1998, date[1] yields ‘‘Nov’’, and date[2] yields 19.

- The following code illustrates two list constructions, which construct a homogeneous list and a heterogeneous list, respectively:

```
primes = [2, 3, 5, 7, 11]
years = ["unknown", 1314, 1707, date[0]]
```

  Now primes[0] yields 2, years[1] yields 1314, years[3] yields 1998, ‘‘years[0] = 843’’ updates the first component of years, and so on. Also, ‘‘years.append(1999)’’ adds 1999 at the end of years.

- The following code illustrates dictionary construction:

```
phones = {"David": 6742, "Carol": 6742, "Ali": 6046}
```

  Now phones["Carol"] yields 6742, phones["Ali"] yields 6046, ‘‘phones ["Ali"] = 1234’’ updates the component of phones whose key is ‘‘Ali’’, and so on. Also, ‘‘David’’ **in** phones returns True, and ‘‘phones.keys()’’ returns a list containing ‘‘Ali’’, ‘‘Carol’’, and ‘‘David’’ (in no particular order).

Variables, Storage and Control

- PYTHON supports global and local variables.
- Variables are not explicitly declared, simply initialized by assignment. After initialization, a variable may later be assigned any value of any type.
- PYTHON’s repertoire of commands include assignments, procedure calls, conditional (if- but *not* case-) commands, iterative (while- and for-) commands and exception-handling commands.
- However, PYTHON differs from C in not allowing an assignment to be used as an expression.
- PYTHON additionally supports simultaneous assignment.
- For example:

```
y, m, d = date
```

  assigns the three components of the tuple date to three separate variables. Also:

```
m, n = n, m
```

  concisely swaps the values of two variables m and n. (Actually, it first constructs a pair, then assigns the two components of the pair to the two left-side variables)
- PYTHON if- and while-commands are conventional.
- PYTHON for-commands support definite iteration.
- We can easily achieve the conventional iteration over a sequence of numbers by using the library procedure range(*m*,*n*), which returns a list of integers from *m* through *n*−1.

- PYTHON supports break, continue, and return sequencers. It also supports exceptions, which are objects of a subclass of Exception, and which can carry values.

- The following code computes the Greatest Common Divisor of two integers, m and n:

```
p, q = m, n
while p % q != 0:
  p, q = q, p % q
  gcd = q
```

- Note the elegance of simultaneous assignment.
- Note also that indentation is required to indicate the extent of the loop body.
- The following code sums the numeric components of a list row, ignoring any nonnumeric components:

```
sum = 0.0
for x in row:
  if isinstance(x, (int, float)):
      sum += x
```

PYTHON Exceptions

- The following code prompts the user to enter a numeric literal, and stores the corresponding real number in num:

```
while True:
  try:
      response = raw_input("Enter a numeric literal: ")
      num = float(response)
  break
except ValueError:
  print "Your response was ill-formed."
```

This while-command keeps prompting until the user enters a well-formed numeric literal. The library procedure raw_input(...) displays the given prompt and returns the user's response as a string. The type conversion ''float(response)'' attempts to convert the response to a real number. If this type conversion is possible, the following break sequencer terminates the loop. If not, the type conversion throws a ValueError exception, control is transferred to the ValueError exception handler, which displays a warning message, and finally the loop is iterated again.

Bindings and Scope

- A PYTHON program consists of a number of modules, which may be grouped into packages.
- Within a module we may initialize variables, define procedures, and declare classes. Within a procedure we may initialize local variables and define local procedures. Within a class we may initialize variable components and define procedures (methods).
- During a PYTHON session, we may interactively issue declarations, commands, and expressions from the keyboard.
- These are all acted upon immediately. Whenever we issue an expression, its value is displayed on the screen. We may also import a named module (or selected components of it) at any time.
- PYTHON was originally a dynamically-scoped language, but it is now statically scoped.

Procedural Abstraction

- PYTHON supports function procedures and proper procedures.
- The only difference is that a function procedure returns a value, while a proper procedure returns nothing.
- Since PYTHON is dynamically typed, a procedure definition states the name but not the type of each formal parameter. The corresponding argument may be of different types on different calls to the procedure.

PYTHON Procedures

- The following function procedure returns the greatest common divisor of its two arguments:

```
def gcd (m, n):
  p, q = m, n
  while p % q != 0:
      p, q = q, p % q
  return q
```

Here p and q are local variables.

- The following proper procedure takes a date represented by a triple and prints that date in ISO format (e.g., ‘‘2000-01-01’’):

```
def print_date (date):
  y, m, d = date
  if m = "Jan":
      m = 1
  elif m = "Feb":
      m = 2
  . . .
  elif m = "Dec":
      m = 12
  print "%04d-%02d-%02d" % (y, m, d)
```

Here y, m, and d are local variables.

PYTHON procedure with dynamic typing

- The following function procedure illustrates the flexibility of dynamic typing. It returns the minimum and maximum component of a given sequence:

```
def minimax (vals):
  min = max = vals[0]
  for val in vals:
      if val < min:
          min = val
      elif val > max:
          max = val
  return min, max
```

- In a call to this procedure, the argument may be either a tuple or a list.
- In effect it has two results, which we can easily separate using simultaneous assignment:

```
readings = [. . .]
low, high = minimax(readings)
```

- Some older languages such as C have library procedures with variable numbers of arguments.
- PYTHON is almost unique in allowing such procedures to be defined by programmers.
- This is achieved by the simple expedient of allowing a single formal parameter to refer to a whole tuple (or dictionary) of arguments.

PYTHON procedure with a variable number of arguments

- The following proper procedure accepts any number of arguments, and prints them one per line:

```
def printall (*args):
  for arg in args:
      print arg
```

- The notation ‘‘*args’’ declares that args will refer to a *tuple* of arguments.
- All of the following procedure calls work successfully:

```
printall(name)
printall(name, address)
printall(name, address, zipcode)
```

Data Abstraction

- PYTHON has three different constructs relevant to data abstraction: packages, modules, and classes.
- Modules and classes support encapsulation, using a naming convention to distinguish between public and private components.
- A *package* is simply a group of modules. A *module* is a group of components that may be variables, procedures, and classes.
- These components may be imported for use by any other module. All components of a module are public, except those whose identifiers start with ‘‘_’’ which are private.
- A *class* is a group of components that may be class variables, class methods, and instance methods. A procedure defined in a class declaration acts as an instance method if its first formal parameter is named self and refers to an object of the class being declared. Otherwise the procedure acts as a class method.
- To achieve the effect of a constructor, we usually equip each class with an *initialization method* named ‘‘__init__’’; this method is automatically called when an object of the class is constructed. Instance variables are named using the usual ‘‘.’’ Notation (as in self.attr), and they may be initialized by the initialization method or by any other method. All components of a class are public, except those whose identifiers start with ‘‘___’’, which are private.

PYTHON Class

- Consider the following class:

```
class Person:
  def __init__(self, sname, fname, gender, birth):
      self.__surname = sname
      self.__forename = fname
      self._female = (gender == "F" or gender == "f")
      self._birth = birth
  def get_surname (self):
      return self.__surname
  def change_surname (self, sname):
      self.__surname = sname
  def print_details (self):
      print self.__forename + " " + self.__surname
```

- This class is equipped with an initialization method and three other instance methods, each of which has a self parameter and perhaps some other parameters. In the following code:

```
dw = Person("Watt", "David", "M", 1946)
```