

UNIT-V

FUNCTIONAL PROGRAMMING LANGUAGES

Functional Programming Languages:

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:

Def: 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 $f(x) = x * x * x$ for the function cube $(x) = x * x * x$ functions.

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

e. g. $(f(x) = x * x * x)(3)$ 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.

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 Form: $h(f) \circ g$ which means $h(x) = f(g(x))$

2. Construction:

A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter

Form: $[f, g]$

For $f(x) = x * x * x$ and $g(x) = x + 3$,

$[f, g](4)$ yields (64, 7)

3. 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:

For $h(x) = x * x * x$

$f(h, (3, 2, 4))$ yields (27, 8, 64)

LISP –

LISP is the first functional programming language, it contains two forms those

are **1. Data object types**: originally only atoms and lists

List form: parenthesized collections of sub lists and/or atoms

e.g., (AB(CD)E)

Fundamentals of Functional Programming Languages:

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

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

This is called referential transparency

A Bit of LISP:

Originally, LISP was a type less language. There were only two data types, atom and list

LISP lists are stored internally as single-linked lists

- Lambda notation is used to specify functions and function definitions, function applications, and data all 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

Scheme:

A mid-1970s dialect of LISP, designed to be cleaner, more modern, and simpler version than the contemporary dialects of LISP, 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

Primitive Functions:

Arithmetic: +, -, *, /, ABS, SQRT

Ex: (+ 5 2) yields 7

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

- **QUOTE** is required because the Scheme interpreter, named **EVAL**, always evaluates parameters to function applications before applying the function. **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))

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)

5. 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

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

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 a list; otherwise ()

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 ()

4. Numeric Predicate Functions

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

Output Utility Functions:

(DISPLAY expression)

(NEWLINE)

Lambda Expressions

Form is based on notation

e.g.,

(LAMBDA (L) (CAR (CAR L))) L is called a bound variable

- Lambda expressions can be applied

e.g.,

((LAMBDA (L) (CAR (CAR L))) '((A B) C D))

A Function for Constructing Functions

DEFINE - Two forms:

To bind a symbol to an expression

EX:

(DEFINE pi 3.141593)

(DEFINE two_pi (* 2 pi))

2. To bind names to lambda expressions

EX:

(DEFINE (cube x) (* x x x))

Example use:

(cube 4)

Evaluation process (for normal functions):

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

(Special forms use a different evaluation process)

Control Flow:

1. Selection- the special form, IF

(IF predicate then_exp else_exp)

e.g.,

(IF (<> count 0)

(/ sum count)

0)

ML :

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 (See Chapter 4)

- It is strongly typed (whereas Scheme is essentially type less) 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)

- Function declaration form: fun function_name (formal_parameters) = function_body_expression;

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

Functions that use arithmetic or relational operators cannot be polymorphic--those with only list operations can be polymorphic

Haskell:

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

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)

Most Important Features

Uses lazy evaluation (evaluate no subexpression until the value is needed,

Has “list comprehensions,” which allow it to deal with infinite lists

Examples

Fibonacci numbers (illustrates function definitions with different parameter forms)

fib 0 = 1

fib 1 = 1

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

Factorial (illustrates guards)

fact n

| n == 0 = 1

| n > 0 = n * fact (n - 1)

The special word otherwise can appear as

guard

List operations

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]

CAR and CDR via the colon operator (as in Prolog)

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

Examples:

product [] = 1

product (a:x) = a * product x

fact n = product [1..n]

List comprehensions: set

notation e.g.,

```
[n * n | n <= 20]
```

defines a list of the squares of the first 20

positive integers

```
factors n = [i | i <= n,
                n mod i == 0]
```

This function computes all of the factors of its , given parameter

Quicksort:

```
sort [] = []
```

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

```
++ [a] ++
```

```
sort [b | b < x; b > a]
```

Lazy evaluation

Infinite lists

e.g.,

```
positives = [0..]
```

```
squares = [n * n | n <= 100]
```

(only compute those that are necessary)

e.g.,

member squares 16 would return True ,The member function could be written as: member []
b = False member (a:x) b = (a == b) || member x

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
```

Applications of Functional Languages:

- 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 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

Scripting languages

Pragmatics

- 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.

A software system often consists of a number of subsystems controlled or connected by a script.

In such a system, the script is said to glue the sub systems together

COMMON CHARACTERISTICS OF SCRIPTING LANGUAGES

Both batch and interactive use

Economy of expressions

Lack of declaration; simple scoping rules

Flexible dynamic typing

Easy access to other programs

High level data types

Glue other programs together

Extensive text processing capabilities

Portable across windows, unix ,mac

PYTHON

PYTHON was designed in the early 1990s by Guido van Rossum.

PYTHON borrows ideas from languages as diverse as PERL, HASKELL, and the object-oriented languages, skillfully integrating these ideas into a coherent whole.

PYTHON scripts are concise but readable, and highly expressive

Python is extensible: if we invoke how to program in C, it is easy to add new built in function or module to the interpreter, either to perform critical operations at maximum speed or to link python programs to libraries that may only be available in binary form

Python has following characteristics.

Easy to learn and program and is object oriented.

Rapid application development

Readability is better

It can work with other languages such as C,C++ and Fortran

Powerful interpreter

Extensive modules support is available

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.

Variables, storage, and control

PYTHON supports global and local variables.

Variables are not explicitly declared, simply initialized by assignment.

PYTHON adopts reference semantics. This is especially significant for mutable values, which can be selectively updated.

Primitive values and strings are immutable; lists, dictionaries, and objects are mutable; tuples are mutable if any of their components are mutable

PYTHON's repertoire of commands include assignments, procedure calls, conditional (if-butnotcase-) commands, iterative (while- and for-) commands, and exception-handling commands.

Pythons reserved words are:

and assert break class continue def del
elif
else except exec finally for from global if
import in is lambda not or pass
print
raise return try while yield

Dynamically typed language:

Python is a dynamically typed language. Based on the value, type of the variable is during the execution of the program.

Python (dynamic)

```
C = 1
C = [1,2,3]
C(static)
Double c; c = 5.2;
C = "a string..."
```

Strongly typed python language:

Weakly vs. strongly typed python language differs in their automatic conversions.

Perl (weak)

```
$b = `1.2`
$c = 5 * $b;
```

Python (strong)

```
=`1.2`
c= 5* b;
```

PYTHON if- and while-commands are conventional

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).

PYTHON was originally a dynamically-scoped language, but it is now statically scoped

In python, variables defined inside the function are local to that function. In order to change them as global variables, they must be declared as global inside the function as given below.

```
S = 1

Def myfunc(x,y);
```

`Z = 0`

`Global s;`

`S = 2`

`Return y-1 , z+1;`

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

Python procedure

Eg :`Def gcd (m, n):`

`p,q=m,n`

`while p%q!=0:`

`p,q=q,p%q`

`return q`

Python procedure with Dynamic Typing

Eg: `def minimax (vals):`

`min = max = vals[0]`

`for val in vals:`

`if val < min:`

`min = val`

`elif val > max:`

`max = val`

`return min, max`

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

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.

Separate Compilation

PYTHON modules are compiled separately.

Each module must explicitly import every other module on which it depends

Each module's source code is stored in a text file. E g: program.py

When that module is first imported, it is compiled and its object code is stored in a file named program.pyc

Compilation is completely automatic

The PYTHON compiler does not reject code that refers to undeclared identifiers. Such code simply fails if and when it is executed

The compiler will not reject code that might fail with a type error, nor even code that will certainly fail, such as:

Def fail (x):

Print x+1, x[0]

Module Library

PYTHON is equipped with a very rich module library, which supports string handling, markup, mathematics, and cryptography, multimedia, GUIs, operating system services, internet services, compilation, and so on.

Unlike older scripting languages, PYTHON does not have built-in high-level string processing or GUI support, so module library provides it.

Assignment Questions:

1. Explain Functional Programming Languages with examples
2. Explain Logic Programming Languages with examples
3. Explain Scripting Languages with examples