Chapter 1
Introduction and Background

Nature of the Course

There are 4 different views to teach the programming languages course:

1- How to program in several programming languages.

2- Survey of the history and nature of several programming languages.

3- How to implement programming languages.

4- The conceptual issues of programming languages (concepts & paradigms).

We will study the nature of PL, What they can do and what they should do,
Instead of what they are and how to use them.

Simply, we will study the structural issues of PL, in two words:
"Concepts & Paradigms"

- ***Concepts***: The basic structure of PL, syntax, semantics, data types, control structures, …etc.

- ***Paradigms***: the model, an approach or the way of reasoning to solve the problem.

Programming Languages Views

There are 3 different views to consider a PL:

1- Designer: (The inventor of the language)

2- Implementer: (The one who build the compiler or the interpreter)

3- User. (The one writes programs in the language)

The course deals with all 3 views with a little emphasis on (3).

Reasons to study Programming Language Concepts

1. To increase the capacity to express programming ideas.
2. Knowing the structure of a programming language makes it easier to learn and understand programming languages.
3. To increase the ability to design new languages. for example it is known that the if...else structure is ambiguous. This means that the compiler does not know which direction to take when parsing it. In a case like this, the designer must take into consideration ambiguity when putting down production rules
4. It is an overall advancement in computing

What is a Programming Language?

A language is a system of signs used to communicate.

(This definition also includes spoken language). All languages have grammar and vocabulary.

Grammar is how we express a language. It is a specific set of rules(with some exceptions in some cases).

Programming languages are the same, they have a set of rules, called "Production Rules", which represent the grammar of the programming language. The main difference between spoken languages and written languages is that the rules are strict, that is, there are no exceptions to rules.

This leads us to a general definition:

> A Programming language is a system of signs used by a person to communicate with the computer machine.

Or a more specific definition:

> A Programming language is a notational system for describing **COMPUTATION** in **MACHINE READABLE** and **HUMAN READABLE** form.

There are three **Key concepts** in this definition:

1. ***Computation.***

This is what computers about. This is everything that happens in a computer on a low level regardless of the application. Everything we know in programming is eventually simplified into small computational operations(Arithmetic operations).

2. ***Machine Readable.***

There must be an algorithm to translate the programming language code in an *unambiguous* and *finite* way. The algorithm must be simple and straight-forward, and usually takes time proportional to the size of the program. Machine Readability is ensured by restricting the structure of the programming language(syntax) to a *context-free grammar*(CFG) which is a system/model to express the syntax of the programming language. Because of such a system, we can create algorithms for translators in a way that produce something machine readable.

3. ***Human Readable.***

A Very important aspect of a program is to be readable. This began with high-level languages. A Programming Language must provide *abstraction* as Data Abstraction : Which means giving variables and data types such names.

(a) **Data abstraction**:

Simple : such as "integer" or "int" or "char"
Structured : such as arrays or strings

(b) Control Abstraction :

Simple: assignment statement, X = X + 3; meaning:
Fetch memory location X, add 3 to it, and store the result back to X.
All this in one simple statement.

Structured: divide the program into groups of instructions such as,
If…else stmt, case stmt, while stmt, procedures, functions, blocks, …etc.

For a more precise and complete definition of programming languages, A Programming language can be divided into **two parts**:

1. Syntax, or the structure.
2. Semantics, or the meaning.

This is considered a concrete definition of a programming language.

Programming Language Concepts

Syntax

The Syntax is the grammar of the programming language. It describes the different structures such as expressions, statements, and blocks.

The Syntax is formally described using a Context Free Grammar (CFG), which is a set of static algorithms and frameworks.

Semantics

The Semantics describe or gives the Syntax structure a meaning. It is more complex and difficult to describe precisely unlike syntax. For example, the meaning of the "if/else" statement must be programmed correctly by the implementer so that the compiler generates the correct code.

Unfortunately, there is no clear formal to describe Semantics analysis unlike Syntax. However, there is a framework called Syntax Directed Translation(SDT) which is used to express the semantic analysis.

Code -> Scanner(Lexical Structure) -> Tokens -> Syntax analyzer -> Object Code

the Scanner takes the statements and analyzes them, creating tokens, Then the Syntax analyzer takes the tokens and tries to create Syntax structures. If a group of tokens creates a valid expression, it moves to the next set of tokens.

For example, let us look at this small segment of code:

```
if(x!=1)

{ n++;}
```

the tokens in this code would be "if","(","x","!=", “1”,

")","{","n","++",";","}" . This is very important for parsing. After the

Scanner has tokenized the statement in the above section, the

Syntax analyzer first checks:

if(x!=10)

if it is correct, then

it checks n++;

if it is correct, then it checks the whole statement to see if the whole if statement is correct.

Paradigms of Programming Languages

There are 4 paradigms of programing languages

(1) Imperative or Procedural Paradigm

This is called Von-Neumann model of computing which is based on Single bProcessor Sequential Execution of instructions. A programming Language that is based on this model is characterized by:

1. Sequential Execution of Instructions.
2. Using Variables to Represent Memory Locations.
3. Using Assignment Statements to Change the Value of a Variable.

An example of a programming language designed with this paradigm is Pascal and C. This is an example function (Greatest Common Divisor):

```
function gcd(x,y:integer):integer;
Begin
     If (x = y) then
         gcd:=x
     else
          if (x > y) then
               gcd:=gcd(x-y,y)
          else
               gcd:=gcd(x,y-x);
End
```

The Same program in C is:

```
int gcd(int n, int m)
{
   if(n==m){
      return m;
      }
   else{
      if(n>m)
         return gcd(n-m,m);
      else
         return gcd(n,m-n);
   }
}
```

(2) Functional Paradigm

Computation is based on the evaluation or calling functions or application of functions. That is why the language is sometimes called applicated language. A programming Language that is based on this model is characterized by:

1. There is **NO** Notion of Variables or Assignment Statements in this Paradigm.
2. Repetition is not Expressed in Loops, but is Achieved by Recursive Calls.

As an example,
Let us take **LISP** (**LIS**t **P**rogramming) language. In **LISP**, everything is a list. In LISP, a list is defined as:

A List is a Sequence of Things Separated by Blanks and Surrounded by Parenthesis.

An example of lists

```
(+ a b)
```

or

```
(+ 2 3)
```

or

```
(if a b c)
```

which means "if a is true, then the value is b. otherwise , the value is c". Let us see some small programs in LISP:

```
>(defun f(x)
(+ x 1))
>f
>(f 3)
>4
```

or another program:

```
>(defun ff(x y)
(+ x y))
>ff
>(ff 3 5)
```

GCD in LISP would be

```
>(defun gcd(n m)
(if (= n m) n
(if(> n m) (gcd (- n m) m)
(gcd (n (- m n ))))))
>gcd
>(gcd 18 16)
>2
```

Lets Write a LISP program to simulate the function power x^n(where x belongs to r and n is an integer)

```
>(defun pwr(x n)
(if (= n 0) 1
(* x (pwr x (- n 1))))))
>pwr
>(pwr 2 4)
>16
```

(3) Logical Paradigm

This Paradigm is based on symbolic logic. The Program consists of a set of statements that describe what is true about these statements. For example, the Greatest Common Divisor function could be written in a Logical language

```
gcd(u,v,u) :- v = 0.
gcd(u,v,x) :- v > 0,
    y is u mode v,
    gcd(v,y,x).
```

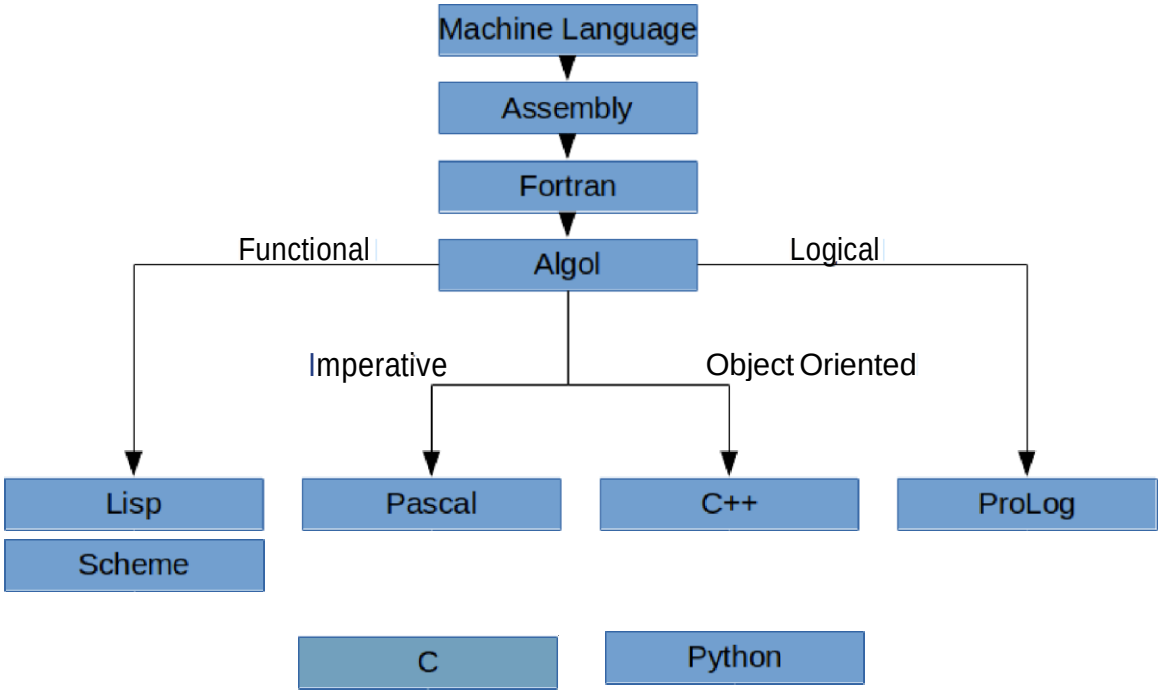
called PROLOG (**PRO**gramming **LOG**ical):

(4) Object Oriented Paradigm

In This Paradigm, the notions of **Object** and **Class** are introduced. It widely spread in the 90's. The advantages of Objected Oriented Programming are:

- Encapsulation of **Data** and **Functions.**
- Inheritance.
- Polymorphism.

The Chart of Language Evolution


Machine Language
Assembly
Fortran
Functional
Algol
Logical
Imperative
Object Oriented
Lisp
Scheme
Pascal
C++
ProLog
C
Python