Reasons to study Programming Language Concepts

1. To increase the ability to learn new languages
2. Knowing the structure of a programming language makes it easier to understand.
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 are its grammar. The main difference between spoken languages and written languages is that the rules are hard ie 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

All 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 in :

1. Data Abstraction : Which means giving variables and such names.
    - Simple : such as "integer" or "int" or "char"
    - Structured : such as arrays or strings
2. Control Abstraction : Which means clarifying operations.
    - Simple : Such as the Assignment Statement "=" or ":"
    - Structured : which is divided to a group of instruction or assignment statements Such as "if/else","for", "while", Block statements.

For a more precise and complete definition of programming languages, instead of a variable definition, A Programming language can be divided into 2 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.

The whole process

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, lets look at this small segment of code:

```
if(x!=10 ){

n++;

}
```

the tokens in this code would be "if","(","x","!=",")","{","n","++",";","}" . This is very important for parsing.

after the Scanner has tokenized the statement in the above section , the Syntax analyser 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 programming languages

Imperative or Procedural Paradigm

This is also called Von-Neumann model of computing which is based on Single Processor 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 Divisior):

```
function gcd(x,y:integer):integer; //int gcd(int x, int y)
Begin
        if(x = y) then//notice that its not ==
                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);
    }
}
```

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 characterised 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, lets take the 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".

Lets 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)
>5
```

GCD in LISP would be

```
>(defun gcd(n m)
(if (= n m) n
 (if(> n m) (gcd (- n m) n)
  (gcd (n (- n m ))))))
>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
```

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

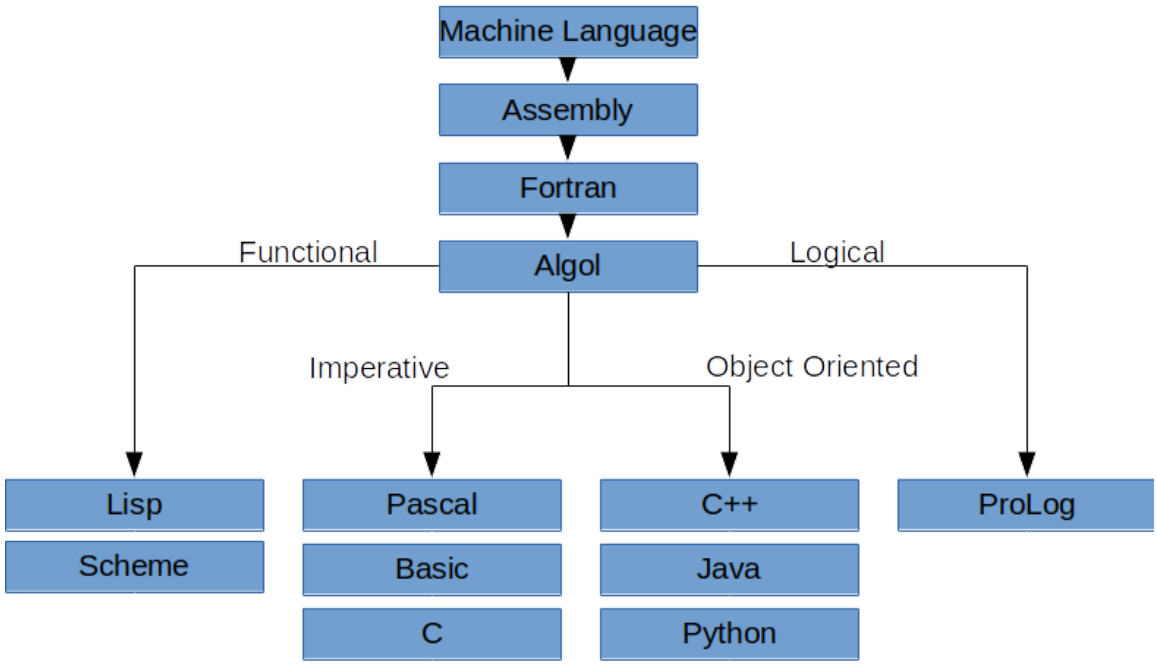
```
gcd(u,v,u) :- v = 0.
gcd(u,v,x) :- v > 0,
    y is u mod v,
    gcd(v,y,x).
```

Object Oriented Paradigm

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

- Encapsulation of **Data** and **Function**
- Inheritance
- Polymorphism

The Chart of Language Evolution


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