Chapter 1
Introduction and Background

Salma
2018 - 2019

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. ✓
↳ developing a translator (compiler)

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 are its grammar. 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***

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).

Source Code → Compiler → object code.
→ Interpreter → intermediate code.

2. ***Machine Readable*** ⇒ develop a translator for the language.

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 (framework) 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.

The syntax (grammar) of PL is expressed by CGF which's a framework to express the syntax of the PL.

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, instead of a variable definition, 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.

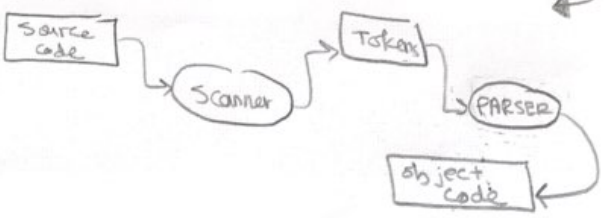
Code -> Scanner(Lexical Structure) -> Tokens -> Syntax analyzer -> Object Code

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

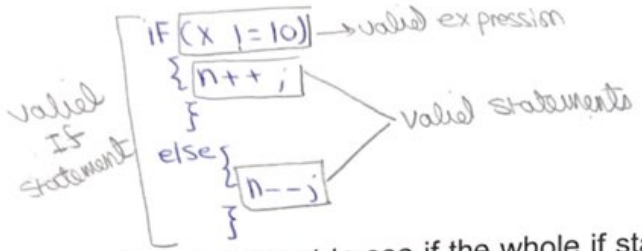
"if","(","x","!=",")","{","n","++",";","}" . This is very important for

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

Imperative or Procedural Paradigm

This is 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 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);
   }
}
```

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 data structure)

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

An example of lists

(+ a b)

or

(a (b c) d)

or (an item of a list which is also a list.)

(+ 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)
> 8
```

GCD in LISP would be (2 codes)

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

```
>(defun gcd(u v)
  (IF (= v 0) u
    (gcd v (mod u v)))))
```

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

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

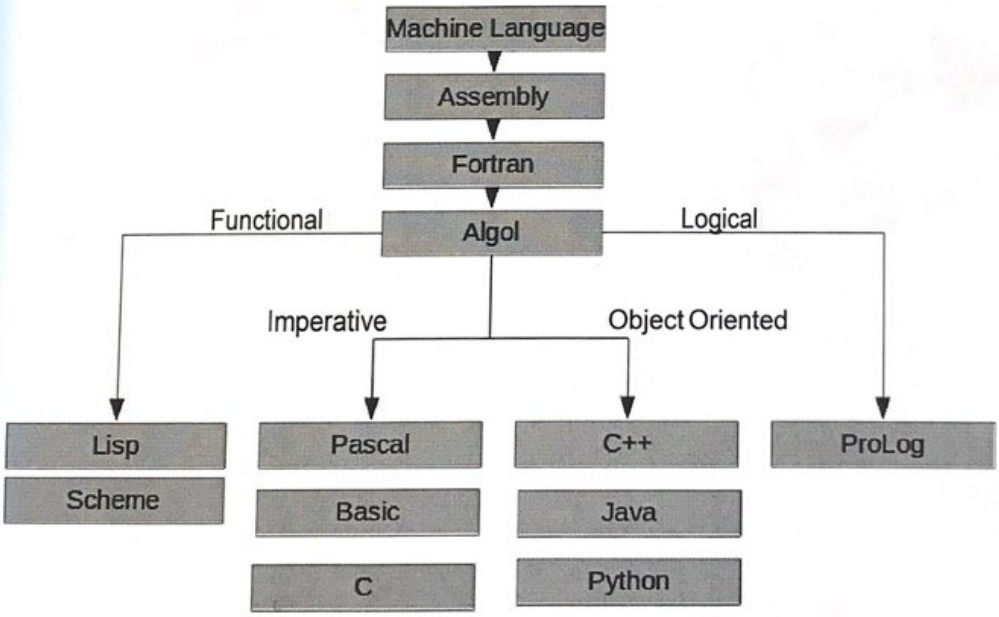
statements that describe what is true about these statements. For example, the Greatest Common Divisor function could be written in a Logical language called PROLOG (**PRO**gramming **LOG**ical):

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 **Functions.**
- Inheritance.
- Polymorphism.

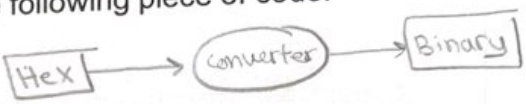
The Chart of Language Evolution



Chapter 3
Programming Languages principles

Language Translation

Early programmers used *Machine Language* to program i.e., The language of numbers. The Programmer wrote his program in HEX which is translated automatically to binary. For example, look at the following piece of code:




```
2 4 63
3 4 46
5 4 57
```

This is a Machine Language program.

Later on, they improved this and created assembly. In compiler construction, there is no difference between Assembly and Machine Language. Assembly simply gave ***mnemonics*** to Machine Language instructions. Assuming some architecture X, the above hex program would be

```
LOD 4,X
ADD 4,y
STO 4,Z
```

in Assembly. Assuming 4 is a certain register, the above code means

Load the contents of memory location 63 whose name is X into register 4.

Add the value stored in memory location 46 whose

name is Y to the value in register 4. Store the value

stored in register 4 to the memory location 57 whose

name is Z.

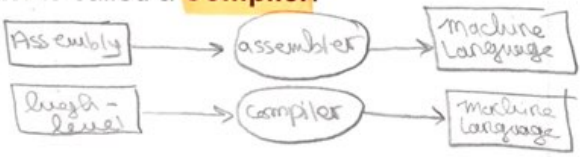
This was still difficult. After Assembly, we created **High Level Languages** such as Pascal, Basic, ADA, C, etc. And with the creation of High Level Languages, there was now a need for **Translators**.

What is a Translator?

The Most general definition of a translator is:

Translator is an Algorithm Which Translates the Source Code Into a Target Code.

If the source code is an assembly language program, and the target code is a machine language program, the translator is called an **Assembler**. If the source code is a high-level language program, and the target code is assembly or machine language, the translator is called a **Compiler**.



Compilers

given the above, a compiler is defined as :

An Algorithm that Translates High Level Language Program to an Assembly or a Machine Language Program.

The Process of compilation and execution, for say, C code is :

```
                                Library Linking          Input
                                      Data V               V
Source Code(*.c) -> Compiler(gcc) -> Object Code(*.obj) -> Executable Code(*.bin) -> Output Data
```

A Compiler generates Object Code (Machine Code).

Advantages:

Generate Object Code
Faster Programs

any Code except machine Language Code needs a translater.

Disadvantage

Complex and hard to implementation.

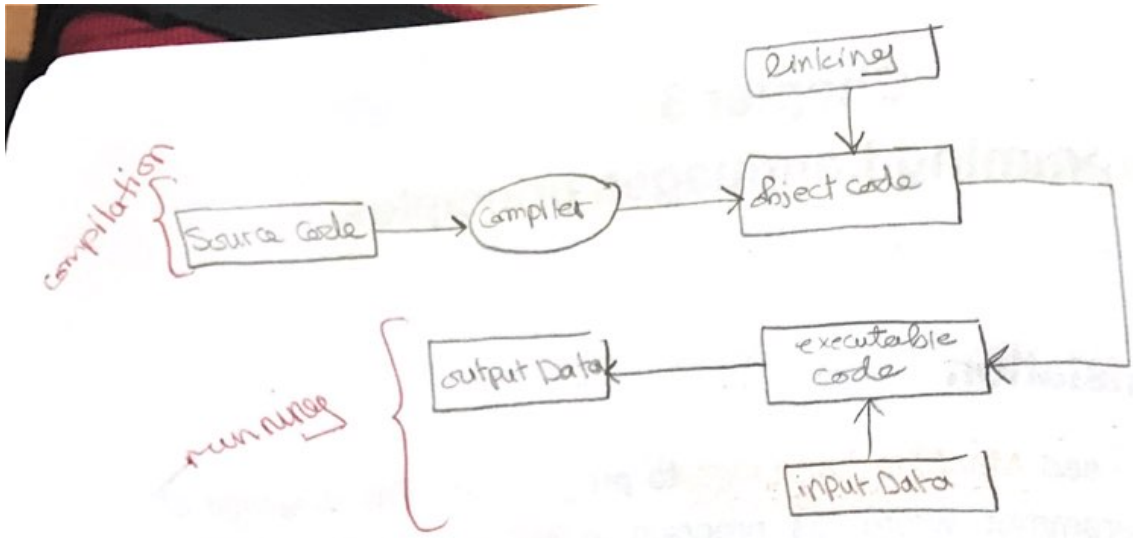
Interpreters

A Simple definition of an interpreter is:

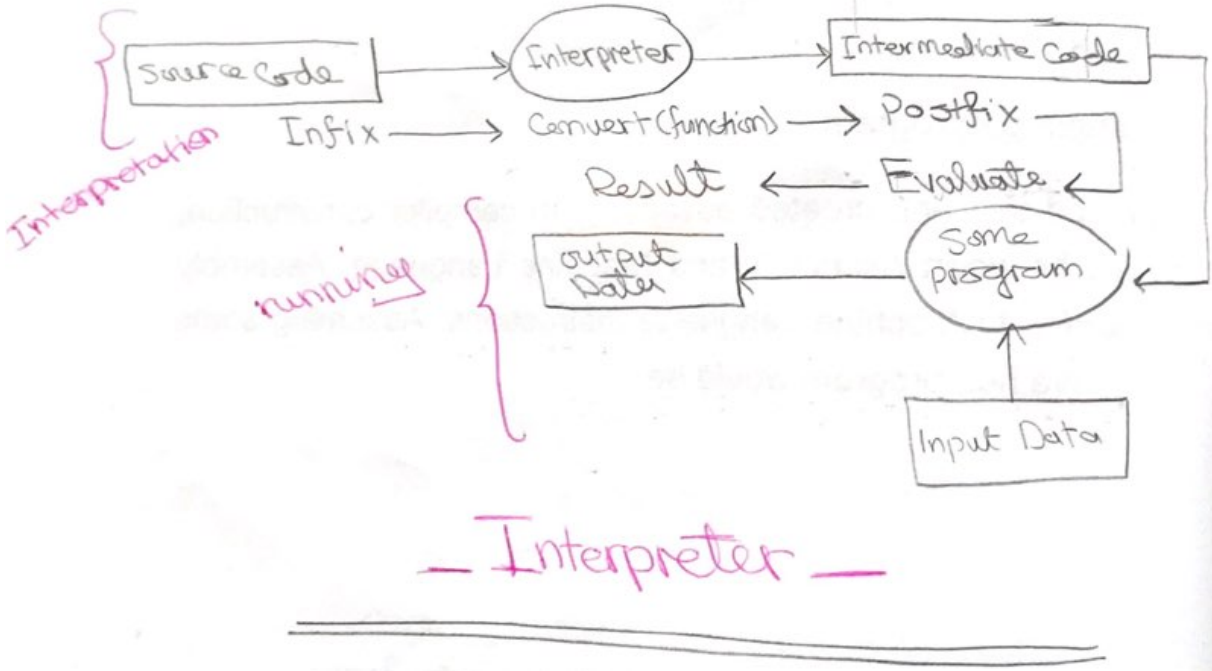
An Interpreter is an Algorithm that Translates the Source Code to an Intermediate Code which is Executed by Another Algorithm(Program) with the Input Data to Produce the Output Data.

The General process of interpretation is :

```
                                    Input Data
                                        V
Source Code -> Interpreter -> Intermediate Code -> Another Algorithm -> Output Data
```



Compiler



Interpreter

A simple interpretation would be changing an equation from infix to postfix and calculation it.

```
                                        Input Data
                                            V
infix Code -> Translator Converter -> postfix Code -> Some Program -> Result
```

in Java :

```
                                  Input Data
                                      V
*.java -> Java Compiler -> Byte Code -> JVM -> Result
```

Advantages

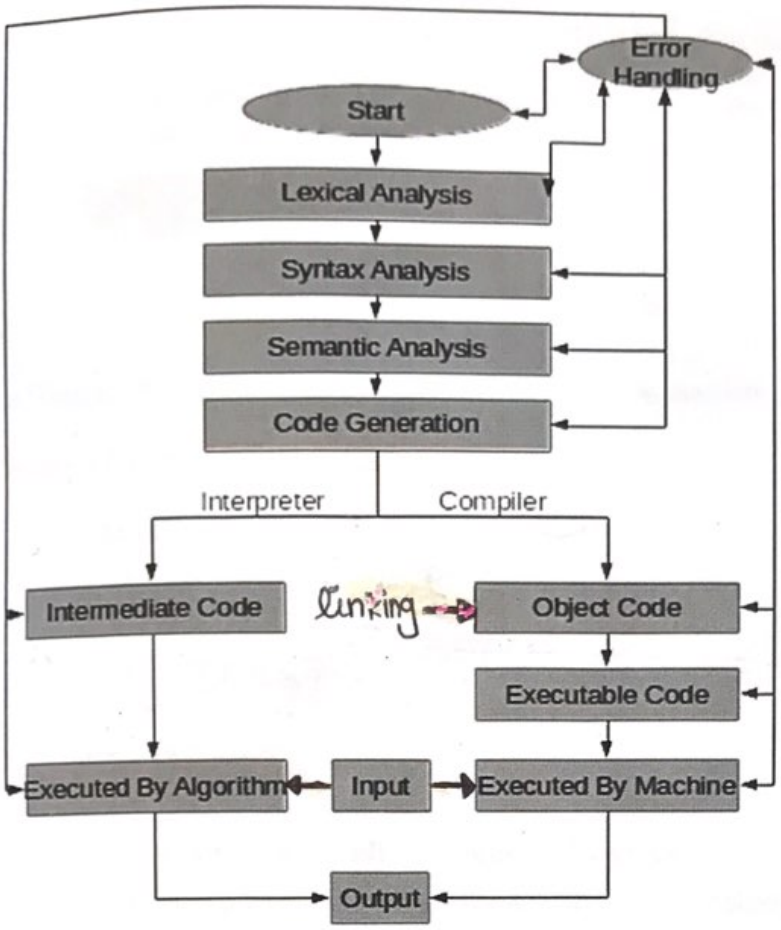
- Interpreters generate a **Portable** intermediate code.
- Easier to Implement

Disadvantages

- Slower

Both Compilers and Interpreters Perform the Following Steps :

- **Lexical Analysis**(scanner) : Which simply groups the characters of the source code to form what is called the **Tokens**. This Only detects legal character errors. example ⇒ (X == 10) (X === 10)
  (, X, ==, 10,) (, X, ==, =, 10,)
  ⚠ NO ERROR ↑ Syntax ERROR!
- **Syntax Analysis** : Groups the set of tokens from the scanner to form Syntax Structures.
- **Semantic Analysis**: Gives the syntax structures meaning. This is the hardest task. ↳ understanding Syntax structures given by the Syntax Analyzer.
- **Code Generation** : Both Compilers and Interpreters do code generation, but they differ in how. While the Compiler generates *Object Code*, the interpreter generates *Intermediate Code*.



These similarities and differences are highlighted in the following diagram:

Even though this flowchart makes it seem that these processes take place one after the other, these processes(lexical analysis, syntax analysis, semantic analysis, etc) are not done independent from each other. Today, almost all compilers are One-Pass Compilers.

Runtime Environment

A Runtime Environment is defined as :

The Space Allocation for Programs and Data in Memory During Execution.

There are 3 types of Runtime Environment :

1. Fully-Static Environment.
2. Fully-Dynamic Environment.
3. Stack-Based Environment.

Fully-Static Environments

In this type of environment, **all** properties of the programming language are predetermined before execution. This means that all the **address allocation is performed when the code is loaded**, not when it is run.

FORTRAN for example, uses this scheme. In FORTRAN, all memory locations of all variables are fixed during program execution. In Addition, FORTRAN has only one type of procedure/function called **subroutine**. In Subroutines, there are **no nested subroutines**, i.e. you cannot define a subroutine in a subroutine. This also means that there is **no recursion**. Thus, the original FORTRAN is suitable for a fully-static environment.

Fully-Dynamic Environments

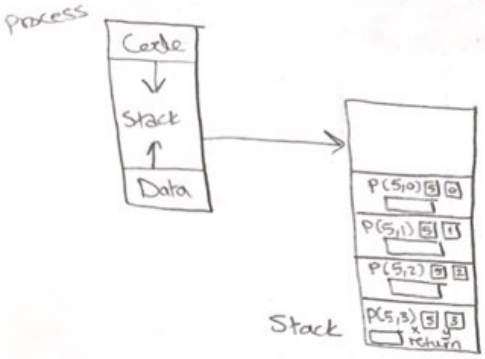
This Scheme is more suitable for dynamically computed procedures such as LISP. It is best with functional and logical programming. This is because it allows us to do recursive function calls, as the allocation is done dynamically.

Stack Based Environments

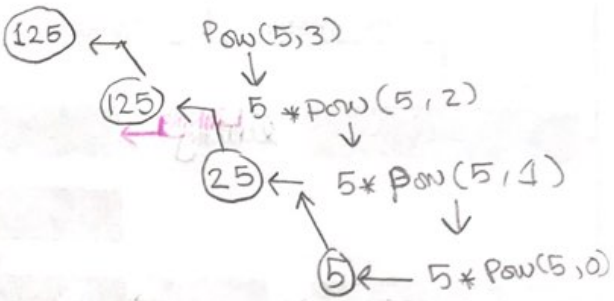
while loading the location of the data is defined.

It Is A hybrid of the above 2 schemes. In This Kind of environment, the static allocation is used for the variables and other data structures, while a stack is used for recursion, nested functions, and procedures during execution. This scheme is best.

* Example on stack-based environments




```
int Pow(int x, int y)
{
  If (y == 0)
    return 1;
  else
    return x * pow(x, y-1);
}
```



The mother of all languages

used with block structured languages (Imperative/Procedural languages) such as all ALGOL-like languages including Pascal, C, Modula, Ada, etc. Most languages today use this scheme.

Languages with strong static structures are more likely to be compiled. i.e., generally, imperative languages are compiled. Conversely, Languages with more dynamic structures are more likely to be interpreted, i.e., generally, functional and logical languages are interpreted.

Error Detection and recovery

During any point or place in the translation process, errors can arise. Generally, efficiency is a trade-off with complexity in error handling. The Faster we handle errors, the less robust our error handling will be. More complex error handling routines, while they do make using the language and fixing bugs easier, they take more processing power and time.

There are 4 types of errors that can arise in the compilation process:

1. Lexical Errors : Lexical Errors arise when an illegal *character* is detected. an example of this is the number symbol in C. they are easiest to find and fix and are detected during Lexical Analysis.
   * Each programming language has its own character set for example, (#) isn't used in PASCAL.
2. Syntax errors : Syntax Errors arise when grammatical errors are detected. This happens when the source code does not follow the grammar of the syntax language, ie, the *Production Rules*. An example of this is missing semicolons in C and Java. They are the a little harder but still easy to find and fix. They are detected during Syntax Analysis
   while (condition) ... while (condition)
   do ↳ Syntax Error X do ↳ Right
3. Semantic Errors : Semantic Errors are detected either during Semantic Analysis or During Execution. There are 2 types of Semantic Errors :
   - Static : And these are pre-execution. An example of these are type mismatch errors.
   - Dynamic : And these are detected **only** during execution. An example of these is division by zero.
4. Logical Errors : These are errors that are related to the logic the code was written in, and what the programmer thinks he means with a statement vs what the compiler actually understands it as. This is completely human error, and is the hardest to fix.

Programming Languages Domain

Programming Languages are divided into several domains

1. Scientific Domain : This Domain includes all applications with a computational base. Languages in this domain include FORTRAN, C, and ALGOL60. This is where programming started originally.
2. Business Domain : This Domain includes all applications used for commercial purposes. Languages in this domain include COBOL(and Database languages) and JAVA. This came afterwards when businesses, especially banks, found use for programming.
3. Artificial Intelligence Domain : This Domain includes languages used for AI. Languages in this domain are LISP and PROLOG.
4. Systems Programming : Which is programming all aspects of the computer (including hardware). Languages in this domain include Machine Language, Assembly Language, and C.
5. Very High Level Languages : These are essentially scripting languages. Languages in this domain include python and bash.

Language Evaluation Criteria : Which Language is the Best?

There is no "best" general programming language. However, we can say that a programming language is *more suitable for a certain application*. There are a lot of factors to consider when we want to choose a programming language.

However, we can compare similar programming languages on certain benchmarks such as speed, space usage, ease of use, libraries available, etc.

Factors that Effect Programming Languages

Readability

It is the most important criteria of programing languages (we made programming languages to be able to understand code after all). It is judging the language by simplicity of which programs can be read and understood, ie, how hard it is to understand a segment of source code. There are several things that contribute to the readability of a language :

- **Simplicity** : a language with a large number of basic components is difficult to learn. Users generally tend to use only some of those features(according to personal preferences). An example of this is how there are many types of loops available(while, for, recursion), but each person has a different affinity to them, or multiplicity (x=x+1,x+=1,x++,++x), where there is more than one way to increment or decrement a variable, and a user only likes to use one of them.

- **Orthogonality** : Orthogonality means the symmetry of relationships among primitives combined to form the constructs/controls ie, the language should not behave differently in different contexts. An example of this is in Pascal, the block statement in loops **must** start with `BEGIN` and end with `END` like this :

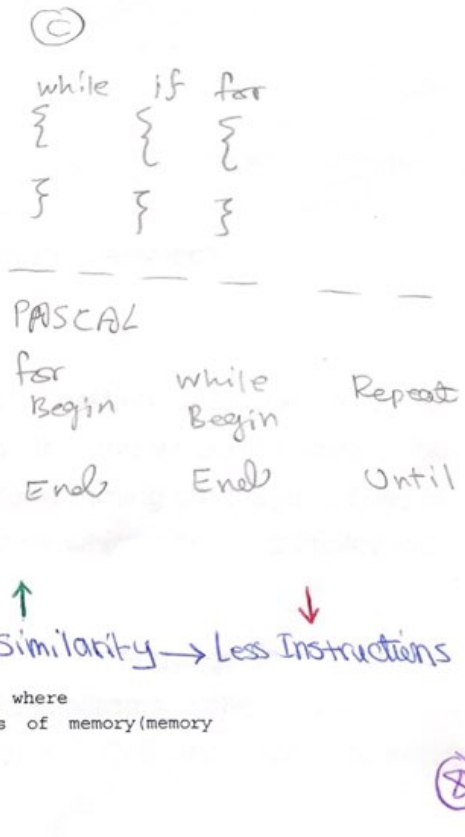
```
```PASCL
for(...)
BEGIN
...
END
```
except in the ```repeat``` statement, which uses

```PASCAL
REPEAT
...
UNTIL
```
or in IBM mainframe, where :

```ASSEMBLY
A Reg1,mem
```
but
:

```ASSEMBLY
AR Reg1,Reg2
```
or in VAX(an OS for mainframe digital corporation), where there is only 1 add instruction for all types of memory(memory locations and registers) :

```ASSEMBLY
Add op1,op2
```


```

```
...
In this case, VAX is said to be more
**Orthogonal**. in short:
> The Less the Orthogonality, The More Instructions There are.
However:
> The Higher the Orthogonality, The More Problems There are to the Compiler.
```

- **Control Structures** : Early Languages such as FORTRAN and COBOL had a limited number of control structures(COBOL had 1 type of loop, the for loop) . As such, the use of the `goto` statement was more prevalent. This caused the language to be less readable. In the 70's, block structured programming languages were introduced as a solution to poor readability.
- **Data Types and Structures** : Sometimes, the use of a datatype can be confusing. Pascal, for example, solved this issue. say we want to have a flag. In pascal, we have the `Boolean` type :

```
flag:Boolean;
...
flag:=true;
if      flag
then
...
```

However, In C, we don't have a boolean type, so if we want to define a flag, we have to use the datatype:

```
int    flag;
flag = 1;
...
if(flag)
...
```

Syntax Consideration :

Identifier length(eg `int` , `for`), separators.

Using Keywords(eg `BEGIN` , `END`) in compound statements.

Writability

Writability is the ability to write programs in a certain language. It is not separated from the readability issue. We can say that the writability issue is the same as the readability issue. Generally, if a programming language is easy to read, it is easy to write and vice versa. The Factors which effect readability also effect writability.

We should compare writability of the programming language in the **same domain**. COBOL is no good for writing a scientific programs, while ALGOL60 is. In the same way, its not a good idea to do AI in ALGOL60 compared, to say, PROLOG.

Reliability

Reliability is how much we "trust" a programming language. A Programing language is said to be reliable if it performs well under all conditions.

The Reliability issue is effected by the following factors :

Type Checking : That is, to check that the operands of a certain operation are of a compatible data type.

Exception Handling : That is, the ability to *detect the error*, *report the error*, and *recover from it*.

Aliasing : That is, Having 2 or more distinct referencing methods , ie, having 2 different names for the same memory location.

Cost

Cost is divided into categories :

Programmer Training. Programmer Training is a function of simplicity and orthogonality. Software Creation. Software Creation is a function of writability.

Cost of Compilation. This means how much time/processing power and space we need to compile the source and create an executable.

Cost of Execution. This means how much time/processing power and space we need to run a program. Cost of Compiler Development.

Cost of Maintenance. This is also a function of readability.

Other Factors

There are also other factors to consider when comparing languages :

Portability : The Ability to move the program and run it on a different platform. This is a huge plus.

Generality : That is, is the programming language a general purpose programming language? Can we use it for everything?

Efficiency : And This includes 3 types of efficiency:

Efficiency in Translation.

Efficiency in Execution.

Efficiency in Writing Programs.