Language Translation

Early programmers used *Machine Language* to program ie, 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 6 3
3 4 4 6
5 4 5 7
```

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:

> A 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 Program 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
```

A Compiler **generates Object Code(Machine Code)**. This is in contrast with an **Interpreter**

Advantages

- Generate Object Code
- Faster Programs

Disadvantages

- Harder to Implement

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
```

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. This means we can **Write Once Run Everywhere**.
- 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.
- Syntax Analysis : Groups the set of tokens from the scanner to form **Syntax Structures**. This Catches syntax errors.
- Semantic Analysis : Gives the syntax structures meaning. This is the hardest task.
- 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.

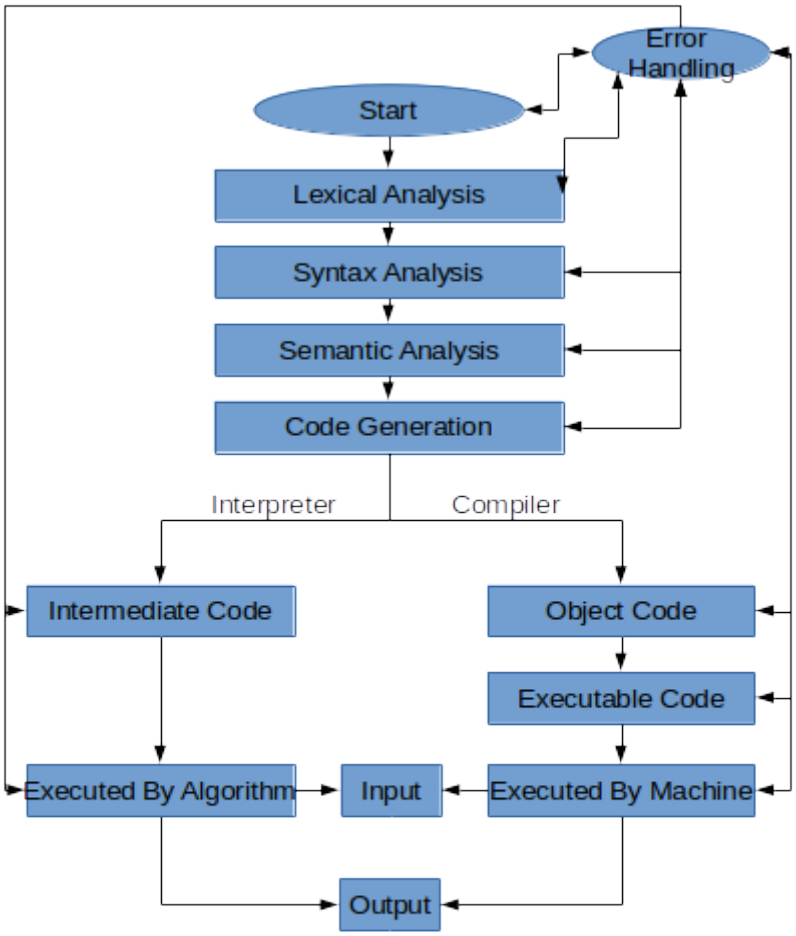

Figure 1: Translation Workflow

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**, ie, 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

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 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. ie, generally, imperative languages are compiled. Conversely, Languages with more dynamic structures are more likely to be interpreted, ie, 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 Lexcical Analysis.
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
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-excecution. 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 :

  ```
  for(....)
  BEGIN
  ...
  END
  ```

  except in the `repeat` statement, which uses

  ```
  REPEAT
  ...
  UNTIL
  ```

  or in IBM maiframe, where :

  ```
  A Reg1,mem
  ```

  but :

`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) :

`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 :

`PASCAL 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 `int` datatype :

`C 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.