# Programming Language a Noob's Perspective

The book arises from my frustration of not finding modern, clear and concise teaching materials that are readily accessible to beginners like me who wants to learn a bit on how to create their own programming language

## What is a langauge

A language is a structured text with syntax and semantics.

## What is a Source Code

A Source Code is structured text written in a programming language needs a translator/compiler to make it executable.

# Elements of Computing

## Instruction Set Architecture (ISA):

- An (ISA) is an abstract model for your computer.
- A **CPU** is an implementation of such ISA.
- A standard ISA defines its basic elements such as data types, register values, various hardware supports, I/O etc. and they all make up the lowest-level language of computing which is the ***Machine Language Instructions.***

## OPcode:

Instructions are comprised of instruction code (aka operation code, in short opcode or p-code) which are directly executed by CPU. An opcode can either have operand(s) or no operand.

*Example*:

- In 8-bits machine where instructions are in 8-bits an opcode. Lets say we want load **0101** binary.
- **Load** might be defined by the 4-bits **0011** following by the second 4-bits as operand with **0101** that makes up the instruction **00110101** in the Machine Language while the opcode for incrementing by 1 of the previously loaded value could be defined by **1000** with no operand.

Since ***opcodes are like atoms of computing,*** they are presented in an opcode table. An example of that is Intel x86 opcode table.

## Assembly Language:

Since it's hard to remember the opcodes by their bit-patterns, we can assign *abstract symbols* to opcodes matching their operations by name. This way, we can create Assembly language from the Machine Language.

In the previous Machine Language example above, **00110101** (means load the binary **0101**), we can define the symbol **LOAD** referring to **0011** as a higher level abstraction so that **00110101** can be written as **LOAD 0101**.

The utility program that translates the Assembly language to Machine Language is called **Assembler**.

## Compiler

Compiler is any program that translates (maps, encodes) a language A to language B. Each compiler has two major components

- **Frontend:** deals with mapping the source code string to a structured format called ***Abstract Syntax Tree (AST)***.
- **Backend(code generator):** translates the AST into the ***Bytecode / IR or Assembly***

Most often, when we talk about compiler, we mean **Ahead-Of-Time (AOT)** compiler where the translation happens before execution. Another form of translation is **Just-In-Time (JIT)** compilation where translation happens right at the time of the execution.

*"If you don't know how compilers work, then you don't know how computers work"*

*"If you can't explain something in simple terms, you don't understand it"*

***Relativity of low-level, high-level***

Assembly is a high-level language compared to the Machine Language but is considered low-level when viewing it from C/C++/Rust. High-level and low-level are relative terms conveying the amount of abstractions involved.

## Virtual Machine (VM)

Instructions are hardware and vendor specific. That is, an Intel CPU instructions are different from AMD CPU. A VM abstracts away details of the underlying hardware or operating system so that programs translated/compiled into the VM language becomes platform agnostic. A famous example is the Java Virtual Machine (JVM) which translates/compiles Java programs to JVM language aka Java **Bytecode**. Therefore, if you have a valid Java Bytecode and Java Runtime Environment (JRE) in your system, you can execute the Bytecode, regardless on what platform it was compiled on.

### Bytecode

Another technique to translate a source code to Machine Code, is emulating the Instruction Set with a new (human friendly) encoding (perhaps easier than assembly). Bytecode is such an intermediate language/representation which is lower-level than the actual programming language that has been translated from and higher-level than Assembly language.

### Stack Machine

Stack Machine is a simple model for a computing machine with two main components

- a memory (stack) array keeping the Bytecode instructions that supports pushing and poping instructions

- an instruction pointer (IP) and stack pointer (SP) guiding which instruction was executed and what is next.

## Intermediate Representation (IR)

Any representation that's between source code and (usually) Assembly language is considered an intermediate representation. Mainstream languages usually have more than one such representations and going from one IR to another IR is called ***lowering***.

## Code Generation

Code generation for a compiler is when the compiler converts an IR to some Machine Code. But it has a wider semantic too for example, when using Rust declarative macro via macro_rules! to automate some repetitive implementations, you're essentially generating codes (as well as expanding the syntax).

## Grammar-Lexer-Praser Pipeline

**Source Code -> Tokenizer -> Lexer -> Parser**

Every language needs a (formal) grammar to describe its syntax and semantics. Once a program adheres to the rules of the grammar in Source Code (for example as input string or file format), it is tokenized and then lexer adds some metadata to each token for example, where each token starts and finishes in the original source code. Lastly, parsing (reshaping or restructuring) of the lexed outputs to ***Abstract Syntax Tree.***

## Grammar

While there are varieties of ways to define the grammar, in this book we will use the Parsing Expression Grammar (PEG).

## Lexer

A lexer, which is also sometimes referred to as a scanner, reads a source program and converts the input into what is known as a token stream. This is a very important step in compilation since these tokens are used by the parser to create an AST (Abstract Syntax Tree). If you are unfamiliar with parsers and ASTs, don't worry! This post is going to focus only on building a lexer.

## Abstract Syntax Tree (AST)

AST comes into picture when we want to go from the string representation of our program like "-1" or "1 + 2" to something more manageable and easier to work with. Since our program is not a random string (the grammar is for), we can use the structure within the expressions "-1" and "1 + 2" to our own advantage and come up with a new representation like a tree.

# Conclusion

- Writing a compiler requires knowledge of a lot of areas of computer science - regular expressions, context-free grammars, syntax trees, graphs, etc. It can help you see how to apply the theory of computer science to real-world problems.
- By understanding how a compiler generates and optimizes code, you will waste less time doing foolish "optimizations" yourself.
- The process of scanning/lexing a file and building a syntax tree out of it is applicable to a much larger set of problems then just building a compiler.
- Helps "generalize" programming languages - once you see that every programming language ends up as machine code in the end, it makes it easier to learn new languages because you see that every language is just a different way of expressing the same basic ideas.

*Some people would counter that there are more useful things you could be doing with your time, like learning a popular programming language or library that could help get you a job (this argument is often used as a reason not to learn assembly language). However knowing how compilers work will probably make it easier to learn new programming languages*

Understanding the process of compilation is a general approach to understanding how the computer works, and therefore provides a broad scope of understanding. Many modern programmers work in complex environments that require little of this understanding, at least on basic levels. An example is java, which hides the linking step of compilation from the developer.

Knowledge is knowledge, it is almost always useful. In my case, I found understanding compilation process exceptionally usefull when doing performance enhancement, which is my job. (I work in a super low latency environment)

If you do all your work in PHP mySql that is great. Just remember that all technologies get outdated, and you will need to understand the next great thing. Having "general knowledge" like understanding compilation provides a conceptional buffer between you and those shmucks who can't adapt.

*Learn how to learn.*