Compiler

Made by: Kazybek Ramazan, Tungatarov Aibol, Amanzholov Arystan, Ansagan Yermakhan

**Introduction**

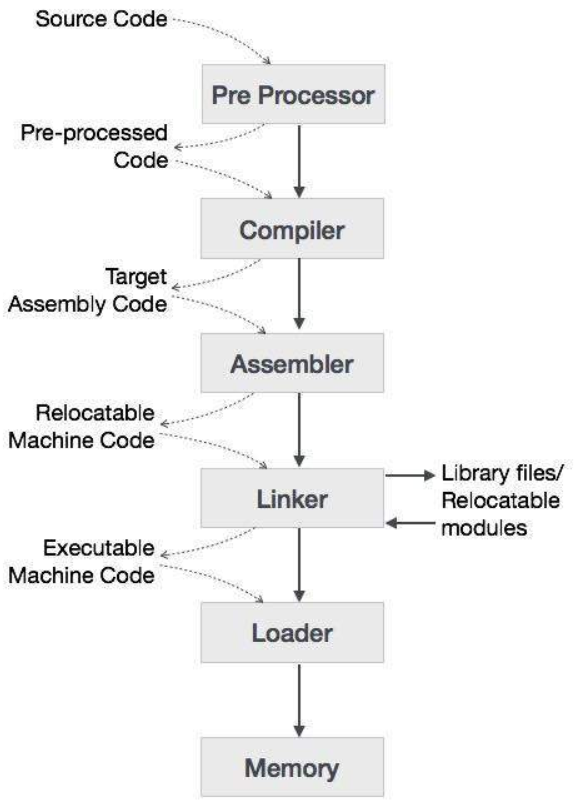
When you talk about compiler, you should start from zero. You have to know what is compiler. What is interpreter? What is the difference between them? And so on…

A compiler translates the code written in one language to another language staying unchanged in its meaning. But it’s also expected that a compiler makes target code taken from source code more efficient and optimized in terms of time and space. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

Computers are a balanced mix of software and hardware. While hardware is just a piece of device and it understands instructions in form of electronic charge and is accepted as on and off. This form people know as binary form (0s and 1s). In order not to write such codes which consist of only binary numbers we have compiler which translates into machine code.

**Language Processing System**

The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. This example is a demonstration of how it’s done. Let you to have written C code and want to run your code.

* The C compiler, compiles your program and translates it to assembly code (low-level language)
* An assembler then translates it into machine code(binary code)
* A linker tool is used to link all the parts of the program together for execution (executable machine code).
* A linker tool is used to link all the parts of the program together for execution (executable machine code).

A compiler can broadly be divided into two phases based on the way they compile.

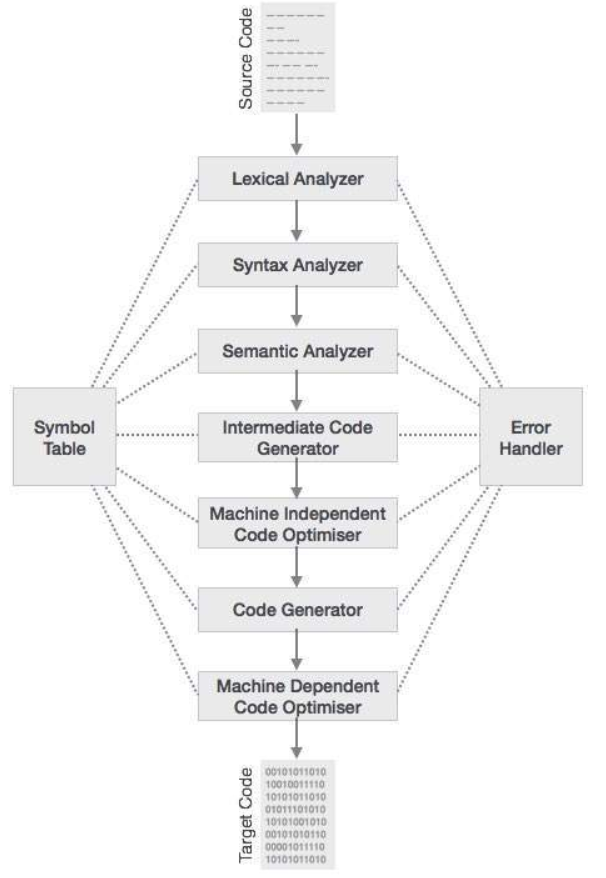
**Analysis Phase**

It’s also known as front end of the compiler. The **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

**Synthesis Phase**

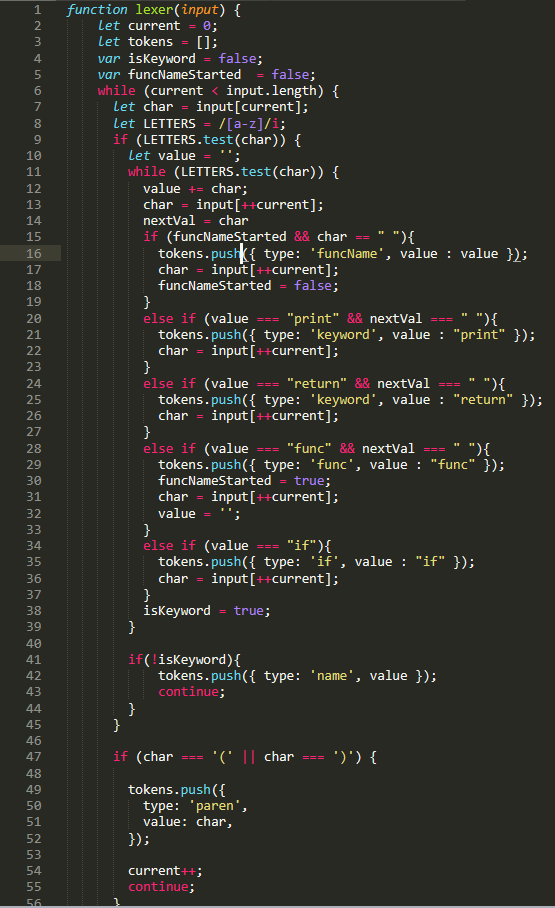
Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.



**Lexical Analysis**

The first phase scans the source code and transforms it into tokens which help to build AST(Abstract Syntax Tree).



**Syntax Analysis**

Also called as parsing, takes the token produced by LA as input and generates a parse tree(AST), also checks the expression made by tokens are syntactically correct.



**Semantic Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions.

### **Intermediate Code Generation**

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### **Code Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.

Should mention that our compiler can make an AST but because we did out manually and didn’t use any libraries it was a bit hard to make full compiler. Our code can only generate only AST, but can’t transform this AST to source code’s AST. Because of that our compiler doesn’t translate source code to target code.



**Regular Expression**

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages.

In our project we used regex while traversing source code but didn’t show all the potential of it.

**Structure of AibolX Language**

Condition: if(condition){

print “any string”

return your\_code

}

As bonus task we wrote it in javascript which means it runs in browser. Give an input into text area and watch json code (CTRL+SHIFT+I)

**Cites**

<https://www.tutorialspoint.com/compiler_design>

<https://www.youtube.com/watch?v=Tar4WgAfMr4>

<https://www.youtube.com/watch?v=eF9qWbuQLuw>

<https://www.youtube.com/watch?v=LDDRn2f9fUk>

And some hindou youtubers’ videos