Universidad Nacional Autónoma de México

Facultad de Ingeniería

And

The alphaX team introduces

AlphaX Compiler

Developers:
Flores Constantino Diego.
Rojas Castañeda Karen Arleth.

Compilers.
Supervisor: Ing. Norberto Jesús Ortigoza Márquez.

Compiler

A *compiler* is a program that translates a source program written in some high-level programming language (such as Java or in this case precisely, C) into machine code for some computer architecture. The generated machine code can be later executed many times against different data each time. In other words, compiling is the transformation from Source Code (human readable) into machine code (computer executable).

The aforementioned "machine language" is difficult to impossible for humans to read and understand (much less debug and maintain), thus the need for "high level languages" such as C. The compiler ensures that out program is TYPED correct. For example, we are not allowed to assign a string to an integer variable. It also ensures that our program is syntactically correct. For example, "x * y" is correct but "x @ y" is not. It is very important to know that the compiler does **not** ensure that our program is logically correct.

Our development was made based on the following process division:

- First of all, it's important to make the analysis of the source program, checking its validation, generating the AST (abstract syntax tree), implementing a lexical analysis phase, all of this to build the complete information to go to the next part of the entire process which is
- Generate the machine code for the specific platform's architecture.

Making this division allows to the complete process being easy to supervise and debug in case of failures, even it leaves an important base to be scalable for another programming languages. Pipeline architecture makes the steps dependents from each other's results, which means the data flow goes through different sequential phases.

Schedule (Phase II – Second Delivery)

Delivery	Date	Activities
Second: Unary Operators	1st Week – from the 14th through the 19th of December.	Team meetings and discussions to identify the necessary changes to complete this second phase; start the proper research to understand how each unary operator works (negation, bit wise and logical negation). Keep the research-learning about Elixir to code as efficiently as possible.

	2nd Week – from the 21st through the 26th of December	Start the changes within the scanner and parser code blocks to improve and adapt the algorithm to the new delivery; first focusing in the negation operator. Understanding the one's complement functioning of each unary operator.
	3rd Week – from the 28th of December through the 4th of January.	Finish the development of the first unary operator (negation) and immediately start analyzing and coding the necessaries parts of the algorithm for the two remaining unary operators
	4th Week – from the 5th through the 8th of January.	Finish the development started in the third week, debug and implement new base and added tests to prove well-functioning of the program. Second release (code) is completed and so is the documentation.

Project Requirements

In this second delivery, the main objective is to compile a program which must be written on C programming language; the structure of the program shall be as the following examples:

```
int main(){
    return -constant;
}
```

```
int main(){
    return ~constant;
}
```

```
int main(){
    return !constant;
}
```

The colors are not random; words in red are reserved words from C programming language, black characters are: open and close parenthesis and open and close braces are those who give to the function a function structure, semicolon ends up the statement, "constant" represents any integer number and finally unary operators are in purple. The output of the entire process must be an executable file.

On the other hand, we have functional requirements:

- The compiler must include an "-h" flag, which is going to deploy the entire option menu with all the available flags for the user.
- The compiler must include a "-t" flag, which is going to show the token list retrieved from the compilation.

- The compiler must include an “-a” flag, which is going to show the Abstract Syntax Tree build from the compilation.
- The compiler must include an “-s” flag, this one is going to generate just the assembler file without an executable file.
- The compiler must include an “-o” flag, this option must include a new name provided by the user for changing the executable file’s name and this way doing it more customizable.
- The execution of the compiler must be able to perform on command prompt (or terminal) based on UNIX systems.
- The executable file must have the same name of the compiled file and must be placed on the same directory.
- The compiler must be developed under Elixir programming language.

And non-functional requirements:

- It is necessary to provide a user’s manual; this must be clear and precise for an easy understanding of anyone who could try to use it.
- Version control using Git-Hub.

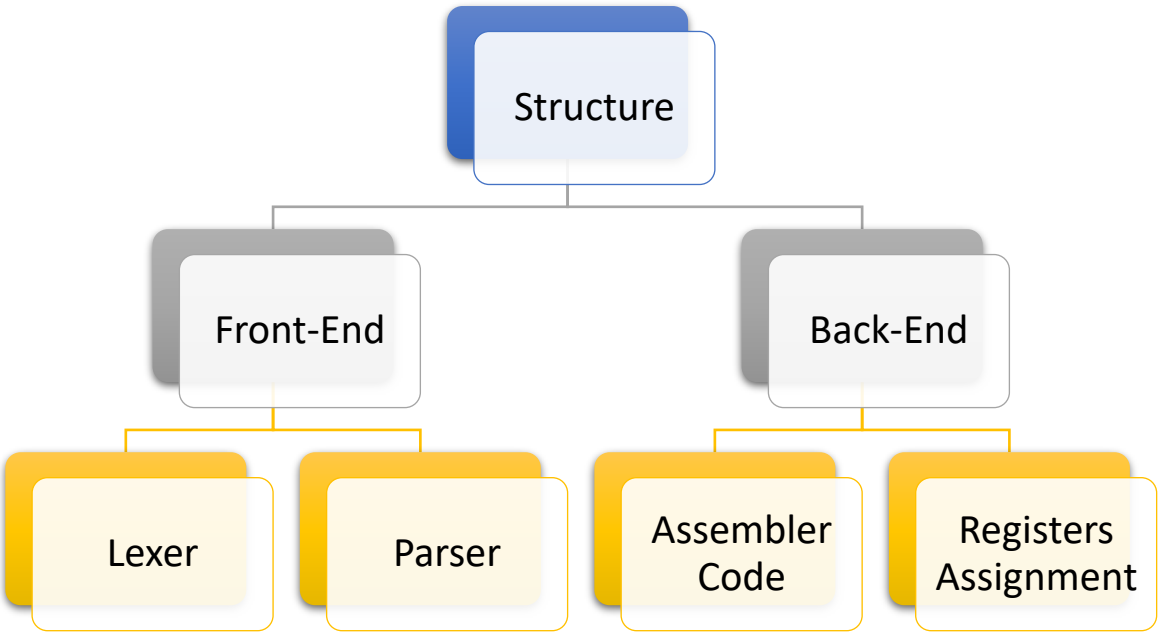
Architecture

As we’ve seen on class, one of the fundamental principles of compilers is that the compiler must preserve the meaning of the program being compiled, this process is well known as the analysis of the semantic. Also, the compiler must ensure the improvement of the input program in a way that it can be easily to understand by the ISA of the processor.

We defined the Compiling process with the following Structure: Source Program (as the main input) this goes to the Front End Block then its output goes to the Optimizer block, its output serve as the input of the Back End block and finally its output becomes in the Target Program.

So, following what has been mentioned our Front-end consists on 2 essential parts: Lexer (which is the lexical analyzer) and the Parser (where the syntax is checked); every process is made based on the C programming language lexical and syntactical rules. In the Back-end part is where the compiler communicates with the computer through the assembler code; here’s where the instruction’s selection and the registers assignment are made.

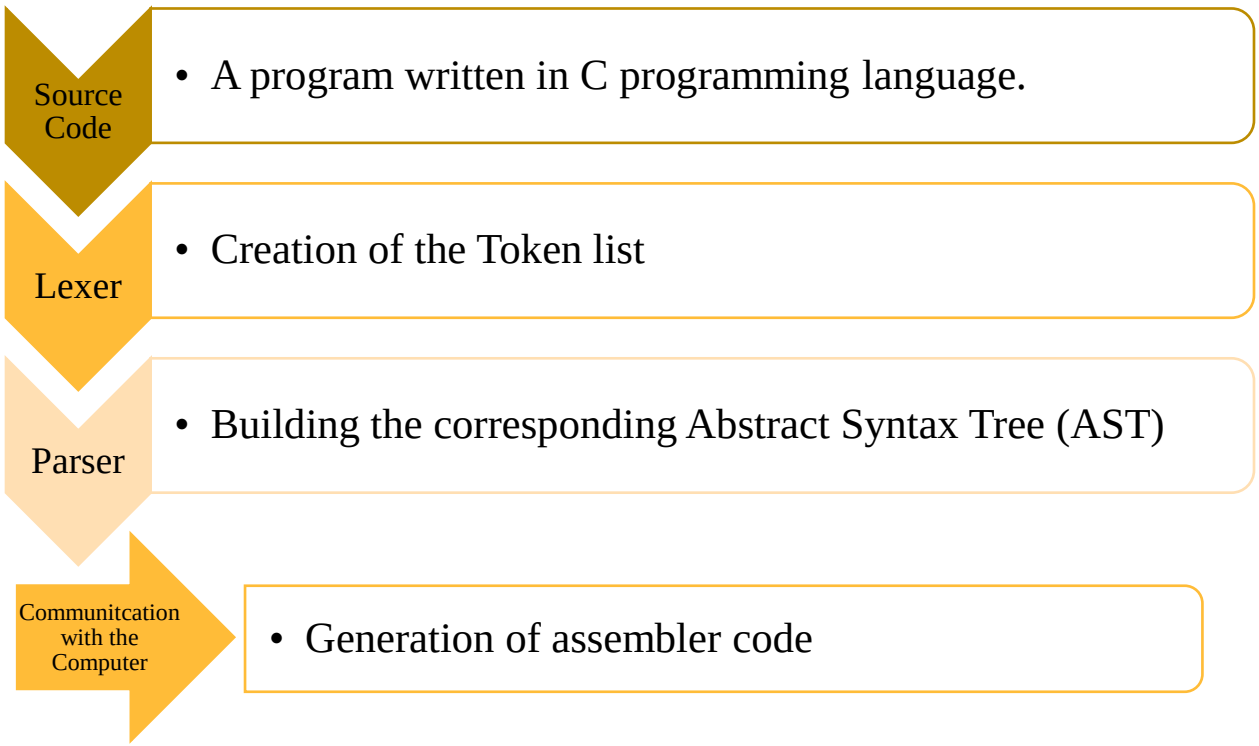
The next diagram summarizes the above established:



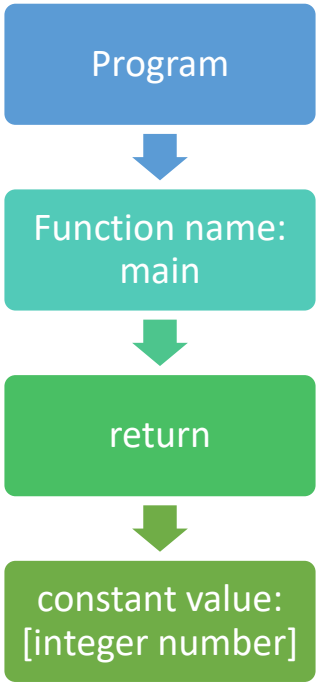
Implementation

At this point of the document it is very clear to notice that our implementation was based on a standard building compiler. At the beginning of the compiling process the first step is to access to the *Lexer*, here's where the lexical analysis occurs verifying that every character between words, parenthesis, braces, the constant (which is an integer number) and the semicolon are typed correctly, in this phase the order of the characters is not important or even if there are not present either; the output of this part of the process is a complete list of each present character on the program, that is called the ***token list.*** The next step is going to happen in the Parser. The input of this one is the *token list*; here's where the syntax of each of the characters included in the token list shall suffer a reviewing process. This review consists of verifying the coherence between characters (that is, the order between them) and the *token list* to be completed, so if any are missing (either reserved word, punctuation mark, etc.) the compilation won't continue. If the Parser does not throw any error, the main process can continue to build the *Abstract Syntax Tree* (AST). This kind of structure allows to easily manipulate the syntactical information of the source code and appreciate the hierarchical relationship that exists between the organization of the content of the program that is being compiled. The final step/phase is to generate the corresponding assembler code which is going to be generated in the programming block called *Linker*. This name is given because this part of the code works as an association among the tokens (human readable) and what the computer can understand (binary code).

Basically, our compiler works as following:



Example of an expected AST



Conclusions

The team concludes that the main objective of this phase of the project was totally covered, this is because we could develop the proper changes and adaptations to our code, that way covering our main purpose which was process three different types of unary operators. We found that every step forward towards the final delivery, if we can reach the proper implementation is going to be easy and easy each time this is due to the solid bases that we are stablishing since the beginning. Once again, we all agreed that the most difficult part was the generation of the assembler code this due to our lack of knowledge of the topic, nevertheless we did the necessary research to get ahead and complement what we have already learned in our past demo. Either way, we realize we are covering the opportunity areas that we have found in the past improving our way to work and making it as well-organized as we could.