COMPILERS

DELIVERY ONE

TEAM HEXACODE

Martínez Vázquez Diego
Reyes Bolaños Victor Daniel
Díaz Guerrero Alan Mauricio
Martínez Hernández Niver Asaid
Ceballos Ricardo Fernando

INDEX

Introduction

During the development of this compiler, we have been working constantly on the making of each of the parts that make this compiler. Of course that our first goal is to make this compiler work for the execution of a code in language C, in the making of this project we will be able to understand the structure of a compiler, get better comprehension of the topics learned in class and analized in the Nora Sandler´s article. During the development of this program we have faced some issues, but also we have learned a lot, like learn a new programming language like Elixir, that represented a very hard challenge for all the members of the team, we can as a team that communication has been a challenge for all the team members, because we didn't have the opportunity to stay all together developing at the same time, our schedules were a lot different.

Project Plan

The global goal while developing this project was to verify that every part of the whole elixir code was able to produce an answer which will be received by the next consecutive part, so it's important to check if it's useful, and foolproof.

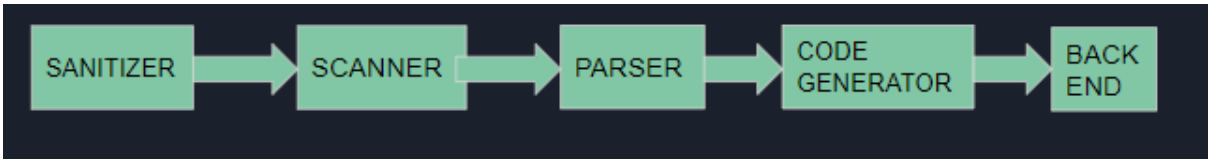
So at the beginning of the course we created a schedule to work around, in which we stipulated to meet on Zoom 3 times a week to discuss many of the doubts we had during the development, most of us were interested in know how elixir actually works, because we weren't familiarized with the coding language.

Compiler design

This project was developed following all the rules that Nora Sandler asks for in her article, we also tried to follow the way to design a compiler working with the material seen in class and taking as a reference the compiler of demonstration that our teacher has provided us.

We had to develop all the front end, meanwhile the backend will only be used with the gcc.

FRONT END



The following programs are the ones that were used for this first delivery.

Sanitizer

Lexer

Parser

Code Generator

Linker

Nqcc

For the design of this first delivery we followed the production rules seen in class.

```
<program> ::= <function>
<function> ::= “int" "main" "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <int>
<int> ::= (0-9)+
```

Compiler architecture

Our project is planned to be shown in four deliveries, at the first one, we must do a compiler that only works with a integer, that it could be any integer, but for all the elaboration of this project we must understand how the compiler is divided and which are the parts that we are need to include to have a correct development.
On the main structure

How every segment really works:

Sanitizer: It receives the main file on C language, and it deletes every blank space or line break from it, after that it puts the string into a list that will be given to our next phase.

Lexer: The scanner or Lexer is the part of the project in which we process the answer from the sanitizer. Specifically, we receive a list without blank spaces that needs to be filtered to get a list of tokens that the next stage, the parser, needs. In this phase we are looking for all the symbols that are part of our main C code.

Parser: The Parser module receives a list of tokens and at the end of the functions, will create the AST tree

Code Generator: This module receives the AST tree generated in the Parser module and with a recursive algorithm (post-order) will generate an assembly code.

Nqcc: The Nqcc is the main part of the compiler that tells us where the code starts.
Each one of the outputs are in a sequential way to be followed by another one.

Ast: this module defines the structure of the tree, every node will have a name, a value, and two more elements for the left and right node.

Linker: This module worked as a substitute of the Backend in the structure of our compiler. The linker calls the gcc compiler and creates the binary file.

Test plan and test suites

Explaining Lexer Tests:

Having in mind Nora Sandler’s document and her tests we have two global current streams to verify if the tests are logical: invalid and valid cases or codes in c which verify different sintaxis.

Valid Cases: These codes in c are the ones that can be compiled by our system without any mistakes, and return an ending result. Having in consideration that each case presents a variation of our main c code.

- Multi_Digit: A variation of it is that our compiler can detect if the constant has above 2 digits.
- NewLines: Another case is that every token or word is separated by a line break.
- No_newlines: In this variation we can see that there's no line break and the tokens are one next to another.
- return 0: In this simple case the value of 0 is also considered as a constant without mistakes.
- return 2: If the constant is equals two it will also have no problem compiling it.
- spaces: Is the case in which we can identify that each token is only separated by spaces.

Invalid Cases:

- Missing parent: Considering that the first parenthesis is not in the code it will show an error.
- Missing retval: A main problem compiling the code is the case of a non
- No brace: A wrong finalization on the test code is a problem while compiling because it won't understand when it really ends.
- No semicolon: An abssence of semicolon is totally a common mistake when programming in C, so our compiler should detect when one is missing.
- No space: This error will match in the test when there is no space between the "return" function and the constant value of the program in C.
- Wrong case: Capitalizing a word in C is also

Explaining Parser Tests:

Having in mind Nora Sandler's document and her tests we have two global current streams to verify if the tests are logical: invalid and valid cases or codes, the Parser has to do work with the token list and generate an AST tree, here are the tests that have been implemented.

Valid Cases: These codes in c are the ones that can be compiled by our system without any mistakes, and return an ending result. Having in consideration that each case presents a variation of our main c code.

- Multi_Digit: A variation of it is that our compiler can detect if the constant has above 2 digits.
- NewLines: Another case is that every token or word is separated by a line break.
- No_newlines: In this variation we can see that there's no line break and the tokens are one next to another.
- return 0: In this simple case the value of 0 is also considered as a constant without mistakes.
- return 2: If the constant is equals two it will also have no problem compiling it.

Invalid Cases:

- No int keyword: When the word int in the token list is missing.
- No main: It occurs when main is missing in the token list.
- Missing open parenthesis: It occurs when the open parenthesis is missing in the token list.
- Missing close parenthesis: It occurs when the close parenthesis is missing in the token list.
- Missing open brace: It occurs when the open brace is missing the token list.
- Missing return keyword: It occurs when the return keyword is missing in the token list.
- Missing number: it occurs when the constant value is missing in the token list

- Missing semicolon:

Explaining Code generator test:

For the test in the code generator we need to make sure that the code we receive at the end of the function can be taken for different architectures, in this case we do the testing for windows and linux architecture.

Particular cases

Space between return and the constant is one of the error tests we didn’t have contemplated at the beginning and was discovered when we tested the code by placing a "return2" assignment, even when the code its wrong because there is no space, the Scanner and the Lexer identified the assignation as correct and create the list of tokens and the AST tree.

Conclusion containing

Until this first delivery we've explained most of the cases that can affect each stage of our compiler, it's important to detail that the intention of the tests is to find mistakes in the main code stages to make it refuse every possible mistake generated by not following the basic rules of programming in c.

The construction of the compiler is in a way very similar to the demonstration one, but it has also a modifications that can allow us to run the code in a suitable manner, we had to to some little changes in different modules like Lexer, Parser and Code Generator, but the way we improved this compiler to the original is very interesting and all the members of the group can say that we have learned new things, like how to work with Github and how to use the Visual Studio that were tools that gave us a new background for engineering software. We as a team made a great effort to understand all the structures and modules work, so we can say that we are in a certain way ready to keep doing the future deliveries correctly.

References

Sandler, N. (2017). Writing a C Compiler, Part 1 from https://norasandler.com/2017/11/29/Write-a-Compiler.html

Elixir. Elixir programming language from https://elixir-lang.org/

Ortigoza, N. Compilers classes on Tuesday and Thursday from 15:00 - 17:00