ExC Compiler

Team Assembly

A small subset of a C compiler written in Elixir

Compilers
Eng. Norberto Ortigoza Márquez
UNAM's School of Engineering
June 2020

Table of Contents

Team Members[1]

This project was developed with the following roles in mind:

- **Customer:** *Eng. Norberto Ortigoza Márquez*
  Defines the compiler features as well as the expected results.

- **Domain Expert:** *Eng. Norbert Ortigoza Márquez*
  Helps explain the compiler concepts, characteristics and the best way to interpret them.

- **Project Manager:** *Néstor Iván Martínez Ostoa*
  Sets the project schedule, holds weekly meetings and maintains a project log to make sure the project deliveries get done on time.

- **Architect:** *Mario H. Garrido Czacki*
  Defines the compiler architecture, modules and interfaces.

- **System Integrator:** *Enrique Hernández Zamora*
  Defines the system development, platforms and tools, the integration environment and a build file to ensure the components work together.

- **System Tester:** *J. Alejandro Ramírez Bondi*
  Defines the test plan and tests suites. Each team member is expected to execute the test suites as the compiler is being developed to make sure the compiler meets the language specification.

[1] Although each team member has a specific role, all of the team members programmed the compiler.

Project Introduction

The project was conceived to develop a small subset of the C compiler using Elixir; a functional programming language. While doing the project, each team member was expected to familiarize with the technical concepts of a compiler, as well as a complete understanding of how each element of a compiler works.

Our Compiler

In our case, we wrote a compiler that takes C programs as inputs and generates executable files. In a nutshell, our compiler works with the following modules:

1. **Reader**: reads the source code in C and provides a useful output that serves as input for the Lexer.
2. **Lexer**: receives the source code and outputs a list of all the tokens contained inside the source code.
3. **Parse**r: receives the list generated by the Lexer and outputs an Abstract Syntax Tree (AST).
4. **Code Generator**: generates the assembly code by traversing through the AST and the specification files.
5. **Code Optimizer**: takes the output of the Code Generator and makes optimizations to the code; in our case, our optimizer focuses on reducing the number of operations in the Code Generator.
6. **Filters**: we added a number of filters in between the main modules to make it easier to catch errors and produce the right output.

It is important to mention that our compiler uses XML specification files to handle most of the architecture structure and grammar. Therefore, the architecture, which we will go into further details in the next sections, is thought so that we only had to modify this files and our compiler would accept the requirements of the project (stages one through four).

Why Elixir?

Elixir is great for this project for the following reasons: data immutability and pattern matching. To develop our compiler we needed a programming language that could work great with independent modules and could easily compare two elements and although we don't use the main features of Elixir and the Erlang's Virtual Machine (BEAM) to handle concurrency; we still benefit from Elixir in ways like pattern matching to compare the output generated by both the Lexer and Parser to the XML specification files.

In addition to the latter, the other main feature we exploited by using Elixir was recursion. We used recursion throughout all our project and although the number of lines is greatly reduced, the complexity of the code increased. Of course, we can't have both but we prefer using recursion since it reduced the lines of code and made or code more elegant.

Project Plan and Scope

Since the complexity of the input C programs will be increased in each stage, planning is key for success. Each deliverable will be divided in four main stages and are explained in Scope section.

Scope[2]

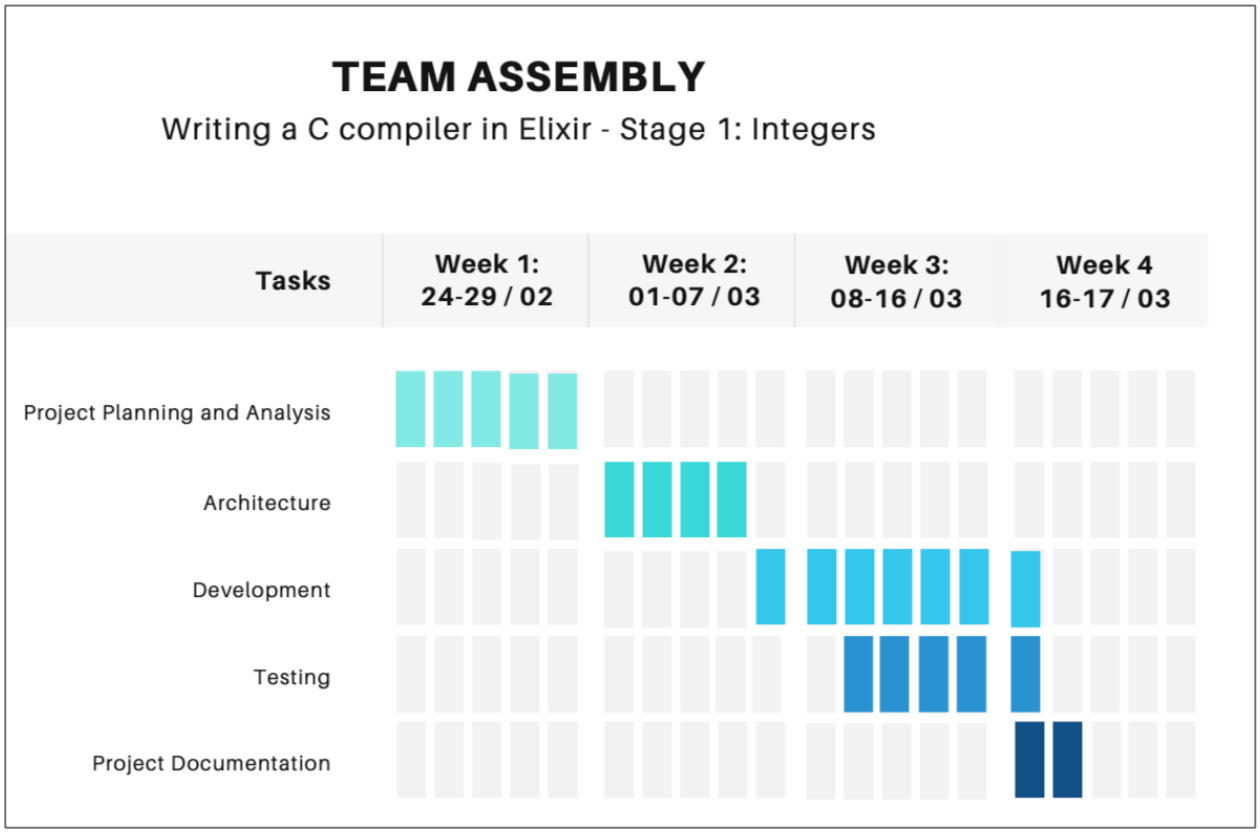
The finished product should be a C compiler that correctly compiles the source code with one of the following expressions inside the main function. Each stage increases the complexity of the compiled expression.

1. Stage One
    a. Integers
2. Stage Two
    a. Unary Operators:
        i. **(-)** Negation
        ii. **(~)** Bitwise complement
        iii. **(!)** Logical negation
3. Stage Three
    a. Binary Operators Part I
        i. **(+)** Addition
        ii. **(-)** Subtraction
        iii. **(*)** Multiplication
        iv. **(/)** Division
4. Stage Four
    a. Binary Operators Part II
        i. **(&&)** Logical AND
        ii. **(||)** Logical OR
        iii. **(==)** Equal to
        iv. **(!=)** Not equal to
        v. **(<)** Less than
        vi. **(<=)** Less than or equal to
        vii. **(>)** Greater than
        viii. **(>=)** Greater than or equal to

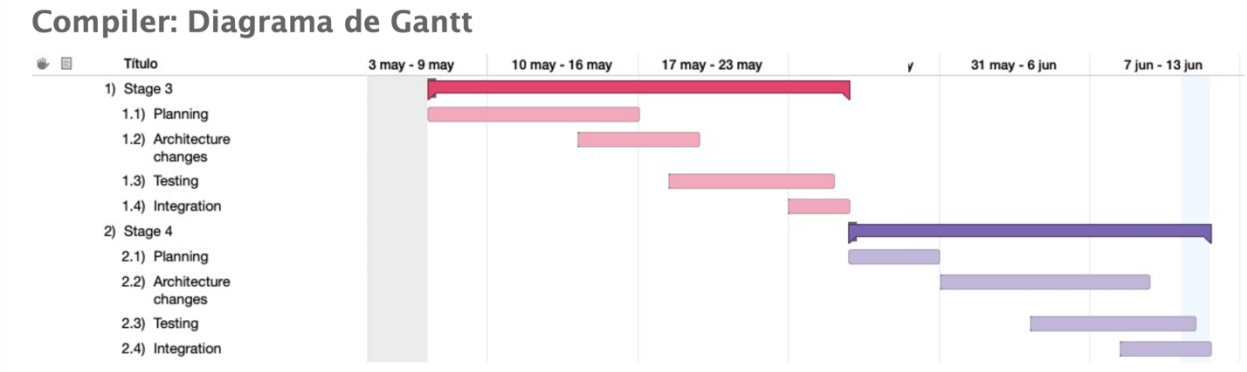
[2] Check Appendix C to see examples of each stage.

Plan

Stage 1



Stages 3 & 4



Compiler Structure and System Architecture

This compiler has been designed to be as flexible as possible. This means that once working, one should not mess with its code. To this end, I have abstracted it to a series of automata (actors) that merely operate over a series of specification files for the desired language. Any modifications or other languages are therefore compatible with this compiler's architecture, thus permitting wide flexibility and easy human modification of the language specification.

Actors

The compiler's basic architecture is comprised of the following actors:

Interface

The Interface's task is receiving the user's commands. An example would be the compilation instruction or the path to the source code one wishes to compile.

Output:

- A string with the pathfile to compile.

Reader

The Reader's task is to load the user's source code into a string. This actor should be able to handle all operations related to file reading.
In addition to the former, the Reader should also be able to load the language's specification files (c_tokens and c_structures) into a generic version of their canonical structures.

Output:

- SCS (Source Code String): A string with the source code file's contents.
- GTL (Generic Token List): A List of tuples which will contain the identifying atom of the token type and its RegEx.
- GAST (Generic Abstract Syntax Trees): A List with all possible Abstract Syntax Tree structures.

Lexer

The Lexer will receive both the SCS and the GTL. This will allow it to tokenize the SCS by matching the RegEx of each token contained in the GTL. If an illegal token is found, the Lexer should notify via an operation status atom (:ok or :error) so the Filter may act.

Output:

- If all tokens are legal:
    - A tuple containing the OTL (Output Token List), a list of the tuples found, and an :ok Elixir atom.
- If there are illegal tokens:
    - A tuple containing the error-inducing token and an :error Elixir atom.

Filter (parser)

The Filter will receive the Lexer's token output. If the operation was done without errors, it will output the OTL. If there were errors, the user should be notified of the offending token, its location and the compilation process should end.

Output:

- If all tokens are legal:
    - OTL
- If there are illegal tokens:
    - Nothing. The user is notified and the program halts.

Parser

The Parser will receive both the OTL and GAST and will then generate an Output Abstract Syntax Tree (OAST) for the program.
Each Token's usage context must be respected. For this purpose, GAST contains all legal production rules for all possible substructures. If there are any context violations, the user should be notified of the offending Token and its position via the Filter and no OAST should be generated.

Output:

- A tuple with the following elements:
    - An elixir status atom (:ok, :token_missing_error or :token_not_absorbed_error).
      :token_missing_error happens if a structure could not be completed because it found an unexpected token, and :token_not_absorbed_error happens if a valid Abstract Syntax Tree could be completed and yet the Token List still had unabsorbed tokens.
    - The OAST (Output Abstract Syntax Tree).
    - The Token List at the end of the compilation.
    - An error cause.
        - If there were no errors, this will be null.

        - If there were errors, it will contain three elements, the structure that could not be completed, the token list that could not be integrated into said structure, and the substructure that was being actually expected.
    - The OTL. This is passed so that the Filter may once again receive it.

Filter (parser)

The Filter will receive the Parser's ouput. If the operation was done without errors, it will output the OAST. If there were errors, the user should be notified of the structure that could not be completed, the offending token, its location and the compilation process should end.

Output:

- If the structure is correct:
    - OAST
- If there are structural violations:
    - Nothing. The user is notified and the program halts.

Code Generator

The Code Generator will receive the OAST in order to generate its Assembly code. The OAST contains its generic (with its registers being placeholders) ASM, so it will merely do a Depth First Search and assign registers so the code may run.
Output:

- AS (Assembly String): An Assembly string representation of the source code.

Code Optimizer

The Code Optimizer will receive the Assembly String in order to optimize it. This step is optional yet important because the compiler's structure requires the result registers of each operation to be indicated by a movq operation. The Code Optimizer will remove these unnecessary operations as they no longer serve any purpose.

- Optimized AS: An optimized Assembly string representation of the source code.

Code Connector

The Code Connector will receive the Optimized Assembly String in order to add concatenate the necessary headers and return instruction so it can run properly.

- AC: Assembly Code ready to be turned into machine code

Writer

The Writer is in charge of receiving the Assembly Code and writing it into a text file for compilation. Naturally, this module should be able to handle exceptions and errors related with file writing.

Output:

- AT: An Assembly text file ready to be turned into object code.

Invoker

The Invoker is the last step in the compilation process, as its task is to pass the pathfile of AT and necessary arguments to GCC in order to generate the final compiled object code.

Output:

- A call to GCC with the necessary parameters to generate the executable.

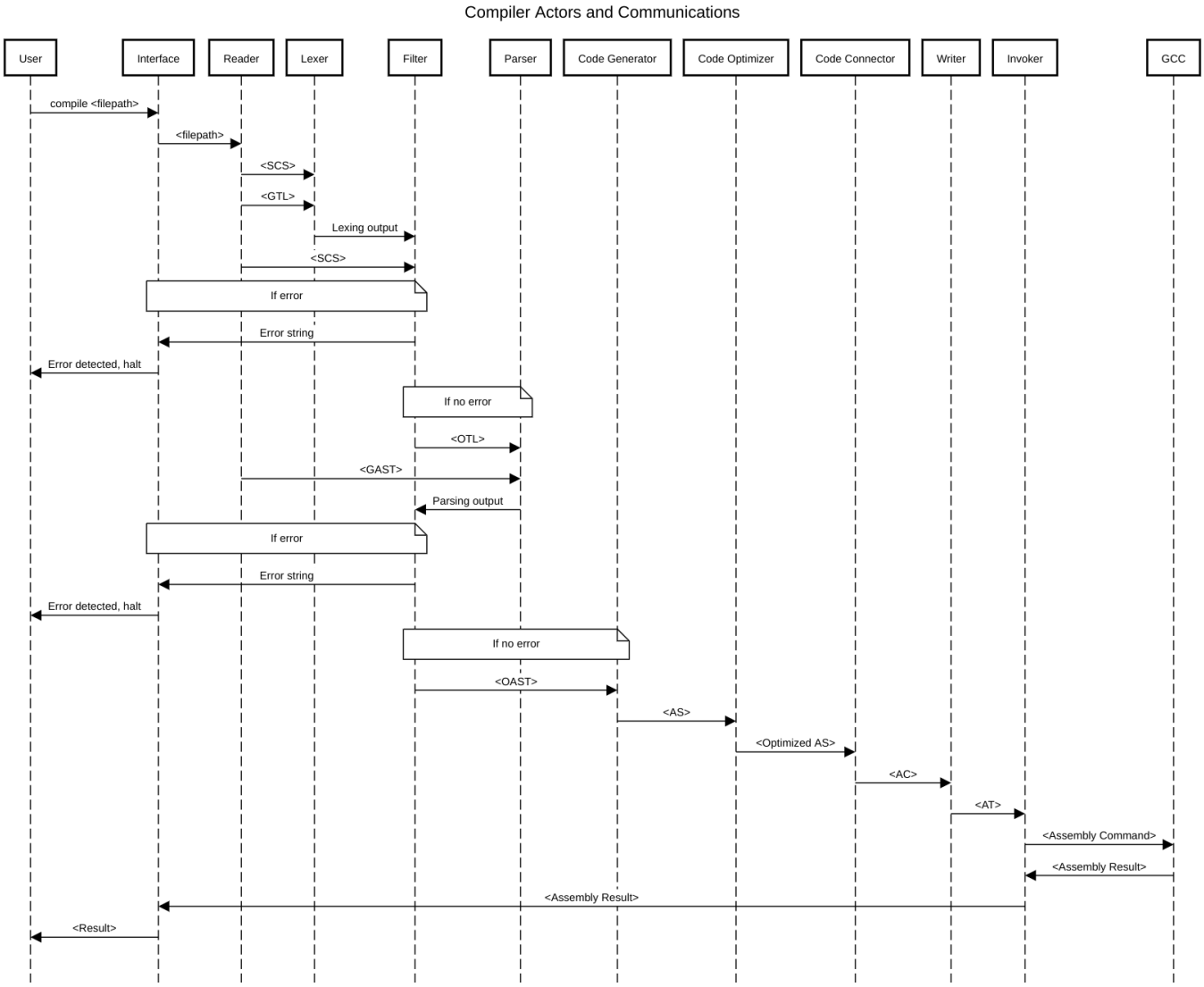


Figure 1: ExC architecture

Data Structures

The following data structures are integral to the communication between actors, and thus they must be well defined:

Token List

A List consisting of Token structs. This represents the tokenized strings and their positions in the document. Tracking the position of the token allows us to trace any errors to the source code and string that caused it.

A Token struct contains:

- Tag: Token's canonical name. E.g. literal, sum, open-parenthesis, etc...
- Expression: Matched string. String that has been matched to the RegEx of the token. This is mainly used to save the value of a literal.
- PosY: Row number of the token in the source code. (Currently unused)
- PosX: Column number of where the token starts in the source code. (Currently unused)

Note that in either the generic version of this list (GTL) or with tokens which possess no variance (if statements, for example) some of these values may very well be empty.

Abstract Syntax Tree List

A List made from Node structs. Node refers to a Node within an Abstract Syntax Tree. A Node struct contains:

- Tag: Structure's canonical name.
- Token: The Token structure that references this part of the structure. This may be empty.
- Children: List with the child substructures (also Abstract Syntax Trees) in the order specified in the GAST. Any non-valid or non-applicable child will be assigned the nil value. If the structure has no children (for example a literal) Children will be empty.
- Class: List with all applicable classes for this Node. This cannot be empty.
- ASM: String file with the equivalent Assembly Language Code.

Specification Files

Specification Files are a crucial component of this compiler. They have the important task of providing all the necessary specifications of the C language and allow us to easily expand upon the compiler without changing any code whatsoever. If at any moment compiling another language was necessary, another set of Specification Files would suffice to implement it in its entirety.

c_tokens

XML File that contains <token> elements with two sub elements:

- tag modifier: Name of the token in its generic shape. For example: plus-sign, if, else, literal, variable, etc...
- <expression>: Regular Expression that matches all possible tokens of this kind.

c_structures

XML File that contains <structure> elements with various sub elements:

- tag modifier: Name of the structure. For example: sum if-statement, evaluation, literal, variable, etc...

- <token>: Token that invokes this structure. If empty, no token is necessary if all substructures are matched (this is helpful for operations).
- Zero or more <substructure>:
    - tag modifier: Name of the substructure. For example in an if-statement: condition, main-body, else-statement. Must be unique within the <structure>.
    - <class>: Contains the names of the <class> tags that apply to this substructure. For example, within the condition of an if-statement: evaluation, operation, comparison, etc...
- One or more <class>: Contain the possible classifications of this substructure. For example, a comparison would have the classes: value, operation, etc...
- <asm>: Contains the Assembly Language equivalent of this substructure.

Project Integration Plan

The following plan describes the integrations we followed to make ExC compiler.

Purpose

This section describes the integration plan of the C language compiler project in Elixir.

Scope

This integration plan shows the necessary software components, in Elixir, that were used to compile the following C program:

```
int main( ) {
return (3 + 4 <= 4 || 1 && 2 != 3 > -6);
}
```

Risk

The greatest risk that can arise is not finishing the project on time, due to the lack of one or more components. The next main risk is to finish the project but does not work adequately or does not meet the requirements.

Complexity

The complexity of the project is high, because between the second and third stages, there is a radical change due to the integration of binary operators. Another factor that increases complexity is the possible case in which the code made in previous stages becomes obsolete.

Maturity

In the first stage, we did not have an advanced level of maturity because none of the project participants were acquainted with Elixir. However, now that we are finishing the project, we are mature enough to use Elixir in other tasks and to continue advancing with the development of ExC.

Starting point

To start developing compiler integrations the following resources were needed:

- Elixir 1.10.2.
- Visual Studio Code (1.42.1) or any other text editor or IDE.
- Revision of *"Write a C compiler"* by Nora Sandler[3] articles.
- Basic Git and Github commands.

How are we integrating?

Since ExC was built based on four different stages and we started from scratch, we needed simultaneous integrations; one per stage. In addition to the latter, we relied on progressive integration and code support since each stage depended on previous installments.

First Stage Integration

1. Reader

   The first integration will allow us to establish the following basic functionality:

   - Read with the **.c** extension and transform it into a string to pass it to the Lexer.
   - Generate two maps, one with atom-expression tuples and the other with possible ASTs. These maps will serve integration three and four and are based on the XML specification files.

2. Lexer

   The second integration will allow us to establish the following basic functionality:

   - Divide the string received by the Reader into a list of tokens.
   - Validate that list of tokens.
   - The output will be a list of atom-string tuples.
   - If there is some error, a list of tuples with the wrong token, column, and row will be displayed.

[3] https://norasandler.com: specifically weeks one through four.

3. Parser

   The third integration will allow us to establish the following basic functionality:

   - Receive a possible AST from the Reader to optimize.
   - Generate an AST with the list of tuples created by the Lexer.
   - If there is some error, it will display a list of tuples with the token generating the error, the column and the row.

4. Code Generator

   The fourth integration will allow us to establish the following basic functionality:

   - Take the AST generated by the Parser to build the code in assembler, from the leaves to the root.
   - The output will be a string with the representative code in assembler.

5. Writer

   The fifth integration will allow us to establish the following basic functionality:

   - Take the string with the assembly language representative code and write it in a text file.
   - The output will be a file ready to be converted into object code.

6. Invoker

   The sixth integration will allow us to establish the following basic functionality:

   - Provide the path and the necessary arguments to gcc to generate the compiled object code.

Second Stage Integration

1. Lexer modification

   The first integration will allow us to establish the following basic functionality:

   - Add the three unary operators (Negation, Bitwise complement and Logical negation).

2. Parser modification

   The second integration will allow us to establish the following basic functionality:

   - The return expression can now take one of two forms: it can be a constant or unit operation.
   - Keep in mind that expressions can contain many unary operators.

3. CodeGenerator modification

The third integration will allow us to establish the following basic functionality:

- Negation and bitwise are achieved with a single instruction in assembler.
  - For the negation the instruction "neg" is used.
  - For the bitwise complement the instruction "not" is used.
- For logical negation, we use the "cmpl" instruction, which is equivalent to the "!" Operator, and the "sete" instruction, which checks if it is the same.

4. Corrections proposed by Domain Expert
   - Change module names from **UpperCamelCase** to **snake_case**.
   - Add documentation in each of the modules.
   - Change the main module to start the program.

Third Stage Integration

1. Lexer Modification

   Each of the operators above will require a new token, except for subtraction – we already have a - token.

   Arithmetic expressions can also contain parentheses, but we already have tokens for those too, so we don't need to change our Lexer at all to handle them.

   The operators to add are:
   - Addition +
   - Subtraction -
   - Multiplication *
   - Division /

2. Parser Modification

   The precedence of the operations for its correct operation must be included. Also, the use of subtraction and negation should be well distinguished, since the same symbol is used, but in a different way.

3. CodeGenerator modification
   The challenge here will be saving the values of each operation along with its result and cleaning the registers when they are used again. In the case of division, the situation is further complicated, because the EDX and EAX registers are treated as the numerator. We will use the "**imul**" and "**idiv**" instructions to support signed operations.

Fourth Stage Integration

1. Lexer modification

   New operators will be treated as a single token even if they have two characters, this to avoid confusion. Although negation has been used in previous stages, the "not equal" should be treated as a single token.
   The operators to add are:
   - Logical AND &&
   - Logical OR ||
   - Equal to ==
   - Not equal to !=
   - Less than <
   - Less than or equal to <=
   - Greater than >
   - Greater than or equal to >=
2. Parser modification

   New operators will be handled the same as in the previous stage, although that involves many levels of precedence.

3. CodeGenerator modification

   Let's handle the relational operators first. Then, for logical operators, conditional jumps are used in assembly language for their correct operation.

4. Addition of Code Optimizer

Greatly compacts the code so it becomes more efficient.

Test Plan and Test Suites

The test plan includes a range of tests that look forward to finding if the compiler produces the adequate results in all the implemented modules. These were divided into two categories: the first ones intended to test the compiler with a valid code; the second ones, intended to verify if the compiler throws the adequate error messages.
These were divided unto the four stages upon which the project was originally divided and on each of the delivery stages. In general, each stage presented a different challenge that was dutifully solved with an iterative approach. In other words, the whole tests were improved step by step, which implied that for stages 3 and 4, all were refactored to accommodate code changes. Thus, we could conclude that the quality and depth of tests for the 4th stage was broader than those of preceding stages.

In general, more than 200 tests were designed and implement amongst the various modules that comprise the ExC compiler. With each stage, code change, new feature implementation or bug fix we were able to determine if the results derived from the compiler were diverging from the scope of the whole project. These would also get us in the direction of developing auxiliary modules to facilitate the development of the tests, which would later turn out useful for further development sessions.

Surprisingly, the act of designing tests and coming up with solutions to the issues encountered after running them, demanded the whole team's involvement. First of all, changes to different modules would have a later impact on the tests that showed the true purpose, as each team member would be guided unto fixing the test in order to minimize the impact on other dependencies.

Test Structure

After finishing the development of stages 2 and 3, we came up with a completely different approach to tests compared to what we had performed during the first stages of development. In the end,

As a whole, the final structure of tests is the following:

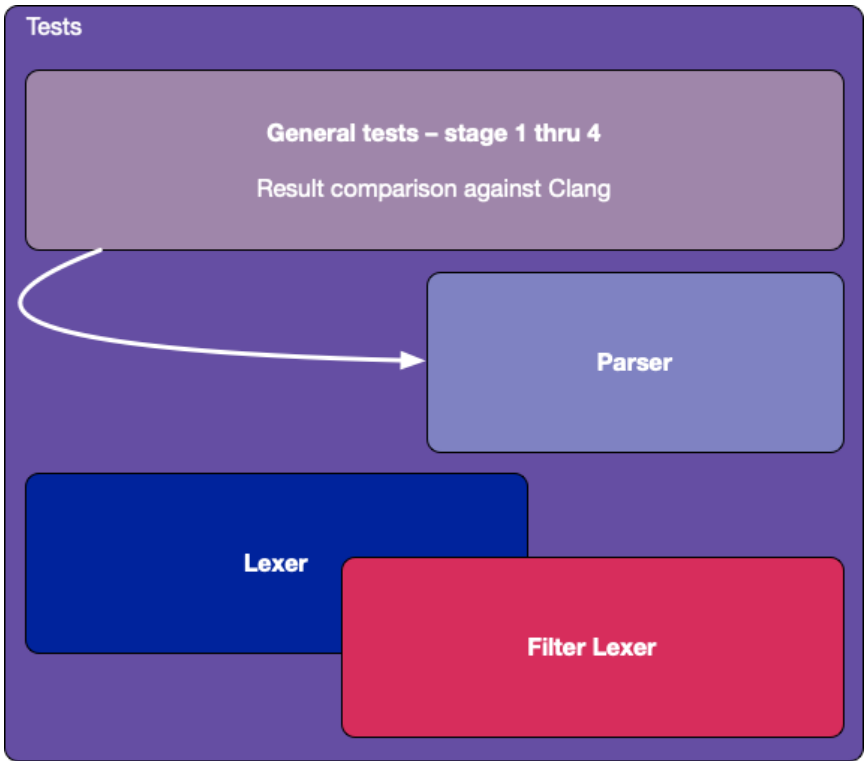


Figure 2: test structure

Project Conclusions

The objective of completing the project with the requirements of all installments described in the stages requirements was met. If there is one general idea that we should consider during all the time we spent developing the compiler, it would be the ease with which we built everything, from the main structure to the testing suite, using Elixir. The whole team agrees that our final implementation has a small amount of iterative code in comparison with other programming languages.

In retrospect, the team believes that the challenges presented with the development of a distinct idea, in contrast with what a typical compiler's architecture is, have brought us a great variety of learning opportunities. Without a doubt, we would accept once again the task at hand of suggesting a customizable architecture for the compiler. Nevertheless, when developing for the best outcome, one must not undermine the theory of the matter at hand. Many of the improvements we achieved after the initial stages came after a better understanding of the theory behind many important concepts surrounding a compiler's internal structures.

As such, we consider that the team was successful in the sense that the initial proposal was maintained during the whole course's development. In other words, despite the challenges we started to set a light on the road to come up with a new and complete solution to create a C compiler.

On the other hand, we are certain that this whole experience gave us the opportunity to better understand what a complete development process, from the plan to maintenance stage, involves. Using tools such as Git and Discord, we were able to communicate and collaborate as a team in order to perform step by step better. It was particularly interesting to face situations upon which we had to give maintenance to certain modules after a new feature was added to the compiler. Thus, the best to which we could implement the basic principles of software engineering would help us to avoid any additional work regarding code refactoring. All in all, this was obtained through communication and documentation, which are normally disregarded as time-consuming.

It goes without saying that this semester was surrounded by external difficulties that could have had an impact on the completion of the project. However, we could say that we coped with them as best as we could and delivered a project that we are proud of. Lessons were learned as a team and personally. The improvements are quite visible when

comparing each and every module and commit in the project's repository. One idea was developed during this whole time and it is extremely gratifying to see the final result.

Appendix

Appendix A: Github repository

The link to the repository is the following: https://github.com/hiphoox/c202-assembly
Note: the installation process of the compiler is located in the README.md file inside the repository.

Appendix B: Hex and Hexdocs documentation

All of our compiler's documentation can be found in the following links:

- ExC Hex's page: https://hex.pm/packages/exc
- ExC Hexdocs' page: https://hexdocs.pm/exc/ExC.html

Inside this pages, the user will find all the information regarding the working process and the main modules we used to build our compiler. We believe this approach was better than just leaving the code commented because it allows the user to take an insight into the module's methods as well as some examples on how to use them.

Appendix C: Scope Examples

Stage one: Integers

```
int main(){
  return 0;
}
```

Stage two: Unary Operators

```
int main(){
  return -7;
}
```

Stage three: Binary Operators Part I

```c
int main() {
  return 10 + 10;
}
```

Stage four: Binary Operators Part II

```c
int main() {
  return 0 || 0;
}
```