A small subset of a C compiler written in Elixir

Team Assembly

Class: Compilers

UNAM's School of Engineering

March 2020

Table of Contents

Team Members

This project was developed with the following roles:

- **Customer:** Norberto Ortigoza Márquez
  Defines the compiler features.

- **Domain Expert:** 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 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:** 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.

Project Introduction

The entire idea of this project is to make a small C compiler using a functional language like Elixir. While doing the project, each team member should familiarize himself 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 will be writing a compiler that takes as input programs written in C and generates output programs in assembly. In a nutshell, our compiler works with the following main modules:

1. Lexer: receives the source code and outputs a list of all the tokens contained inside the source code.
2. Parser: receives the list generated by the Lexer and outputs an Abstract Syntax Tree (AST).
3. Code Generator: generates the assembly code by traversing through the AST.

Why Elixir?

Well, Elixir is great for this project for the following reason: Elixir is all about transforming data. This means that Elixir is oriented, being a functional language, to develop architectures that tend to have inputs that needs to be transformed into an output in a different context.

Elixir is a great language for this project because it allows us to iterate over lists with a reduced number of lines of code, however, this means that the abstract complexity increases because now we have to deal with recursion.

Project Plan and Scope

The entire deliverables of this project will be delivered by stages. Each stage will increase in complexity of the input C source program. Therefore, planning is key for the success of this project.

Scope

1. C Compiler written in Elixir that compiles source programs with one of the following expressions inside the **main** function:

1.1 Stage One: Integers 1.1.1 1, 2, 3, etc.

1.2 Stage Two: Unary Operators

- Negation (-)
- Bitwise complement (~)
- Logical negation (!=)

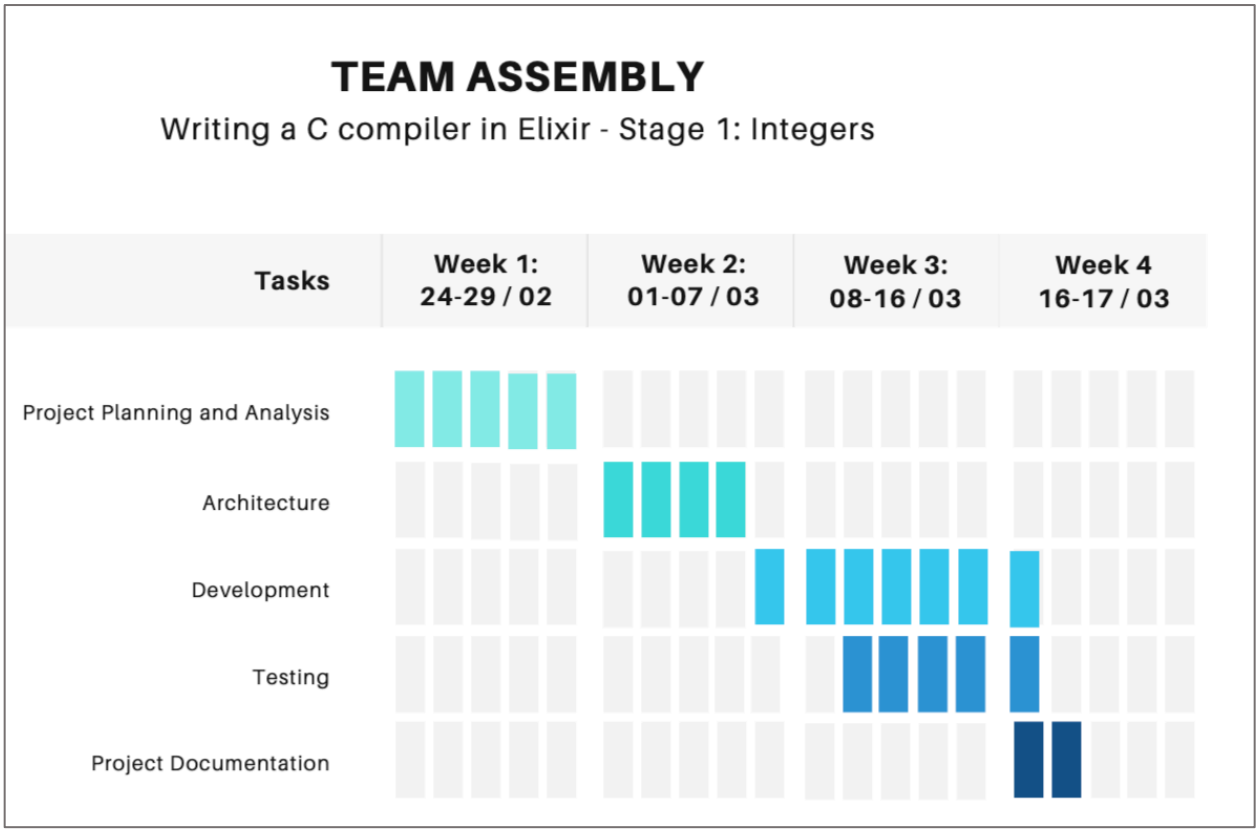
1.3 Stage Three: Binary Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

1.4 Stage Four: More Binary Operators

- Logical AND (&&)
- Logical OR (||)
- Equal to (==)
- Not equal to (!=)
- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

Plan


TEAM ASSEMBLY
Writing a C compiler in Elixir - Stage 1: Integers
Tasks
Week 1: 24-29 / 02
Week 2: 01-07 / 03
Week 3: 08-16 / 03
Week 4 16-17 / 03
Project Planning and Analysis
Architecture
Development
Testing
Project Documentation

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 composed 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: A string with the source code file's contents.
- GTL: A List of tuples which will contain the identifying atom of the token type and its RegEx.
- GAST: 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. For each illegal token found, the user should be notified of the offending token and its position via the Interface and no OTL should be generated.

Output:

- If all tokens are legal:
    - OTL: A Token List generated from the SCS.
- If there are illegal tokens:
    - TEL: A list of tuples containing three elements, the illegal token string, the token's row, and the token's column.

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 Interface and no OAST should be generated.

Output:

- If all tokens are contextually legal:
    - OAST: An Abstract Syntax Tree with the entire program logic.
- If there are context violations:
    - ASTEL: A list of tuples containing three elements, the context violating token, the token's row, and the token's column.

Code Generator

The Code Generator will receive both the OAST and GAST in order to generate the Assembly code for the OAST. The GAST will contain the Assembly equivalents for every substructure, so by processing the leaves of the OAST and going up the tree the AS will be generated.

Output:

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

Writer

The Writer is in charge of receiving the AS 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:

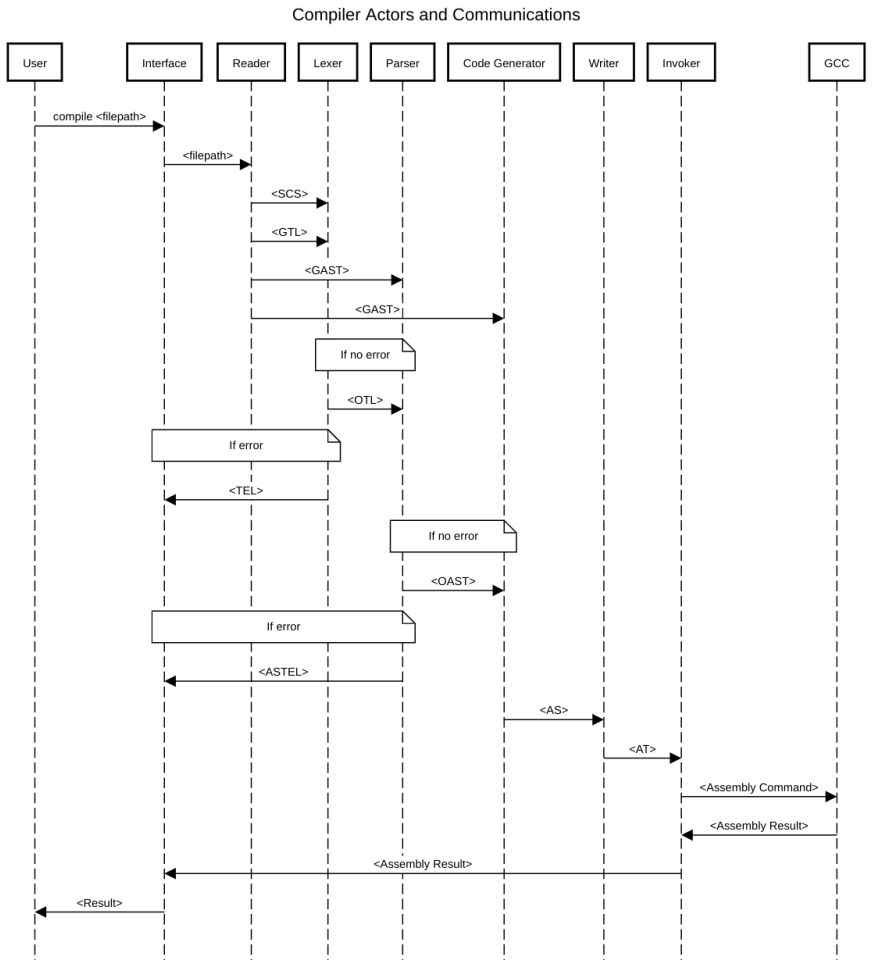
- AT: An Assembly text file ready to be turned into object code.

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.



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.
- PosX: Column number of where the token starts in the source code.

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 integration plan describes the first release of the C language compiler.

Purpose

This document describes the integration plan for the first part of the C Language Compiler project in Elixir.

Scope

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

```
int main( ) {
return 2;
}
```

Risk

The greatest risk that can arise is not finishing the job on time, due to a lack of many or few components.

The next important risk is to finish the project, but that does not work optimally or does not meet the stated requirements.

Complexity

The first part of this project is not very complex, but it is important to keep in mind in order to obtain a greater maturity in the next installments, as well as to reduce their complexity.

Maturity

In this first installment, we still do not have an advanced level of maturity (but it is sufficient) because none of the project participants has used the Elixir programming language before.

Start point

To start developing compiler integrations the following resources will be needed:

4. Installation of the latest version of the Elixir language (version 1.10.2).
5. Installation of the latest version of Visual Studio Code (version 1.42.1) or failing any other text editor or IDE.

6. Review and read part 1, Write a C compiler, by Nora Sandler (web address attached to the references).
7. Management of basic git commands.
8. Review of the Github **nqcc elixir** repository provided by the teacher (web address attached in the references).
9. The project will start from scratch, but with the support of the teacher's repository.

Integration mode

We will use simultaneous implementation because we will start the project from scratch, but we will also base ourselves on progressive implementation, that is, we will use the provided repository to support us, because the maturity we have in this first installment is only enough.

Integrations

Integration one (Reader)

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

- Read the file in .c format 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.

Integration two (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.

Integration three (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.

Integration four (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.

Integration five (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.

Integration six (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.

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.

Project Conclusions

The objective of completing the project with the requirements of this first installment was met, although there were some time problems during development, we managed to complete it. If there is one idea we should take of out this first stage, it would be the ease with which we built the compiler using Elixir. We all agree and are happy that our final program has very little iterative code in comparison with other programming languages.

Appendix

Appendix A: Github repository

The link to the repository is the following: https://github.com/hiphoox/c202-assembly

Appendix B: How to install and run the compiler

The process to install and run the compiler can be found in the following image. Make sure to open a terminal and follow the commands shown in the image.

Figure: how to install, run and clone the compiler.