Section I – General Information	
Project Name:	Building a C compiler in Elixir
Project Sponsor:	Norberto Ortigoza Márquez
Email account:	norberto@bunsan.io
Section II – Details	
Introduction:	Compilers are at the core of computer science, everything in the world right now is technology because computers are amazing at automatizing processes and making our lives more efficient, however, the bridge between our world and the computer are compilers. Compilers are computer programs responsible for translating from a high-level language to assembly language (although there are exceptions). Every compiler has to two main principles: 1. *The compiler must preserve the meaning of the program being compiled.* 2. *The compiler must improve the input program in some discernible way.* Each compiler is composed of the following basic components: • Lexer/Scanner: responsible for transforming the source code program into a list of tokens, i.e., all the written components a source code program has. • Parser: responsible for analyzing syntactic structure; if there aren't then the parser generates an Abstract Syntax Tree with the list of tokens. • Code Generator: responsible for analyzing semantic structure, if there isn't any error, then it produces the assembly code (although it could be any other high-level language) • Optimizer: responsible for detecting syntactic and semantic holes that could be optimized by writing the source code in a better way. Now, compilers are extremely complicated programs that can contain millions of lines of code, and since we are in an introductory course, we will focus on building a compiler that recognizes extremely simple programs written in C. Much of these programs are programs that have a defined structure such as the following: *int main () { return <expression>; }* Where <expression> will be substituted with one of the following possibilities: • Integers

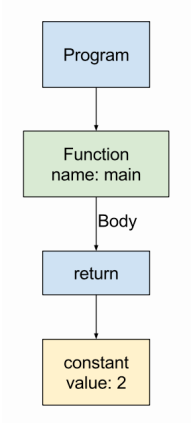
	• Unary operators • Binary Operators The compiler will be written in Elixir, a functional programming language that focuses on parallel computing.
Project purpose:	Learn how to build a compiler and identify its main parts.
Specific objectives:	Compiler that can compile C programs in a 64-bit architecture with one of the following structures defined in the **Introduction**.
Detailed Scope:	1. C Compiler written in Elixir that compiles source programs with one of the following expressions inside the **main** function: 1.1 Phase One: **Integers** 1.1.1 1, 2, 3, etc. **1.2** Phase Two: **Unary Operators** 1.2.1 Negation (-) 1.2.2 Bitwise complement (~) 1.2.3 Logical negation (!=) **1.3** Phase Three: **Binary Operators** 1.3.1 Addition (+) 1.3.2 Subtraction (-) 1.3.3 Multiplication (*) 1.3.4 Division (/) **1.4** Phase Four: **More Binary Operators** 1.4.1 Logical AND (&&) 1.4.2 Logical OR (||) 1.4.3 Equal to (==) 1.4.4 Not equal to (!=) 1.4.5 Less than (<) 1.4.6 Less than or equal to (<=) 1.4.7 Greater than (>) 1.4.8 Greater than or equal to (>=) 2. The compiler should contain the following modules: 2.1 Scanner 2.2 Parser 2.3 Code generator 2.4 Optimizer (optional) 3. Project Documentation: 3.1 Project Charter 3.2 Project Scope 3.3 Installation guide
Functional Requirements:	0. Common to all phases 0.1 Compile the source file **0.1.1 ./assembly source.c** 0.2 Assemble it into an executable 0.2.1 **./gcc -m32 source.s -o source** 0.3 Run the executable 0.3.1 **./ source** 0.4 Check the return code 0.4.1 **echo $?** **output of source.c** 1. Phase One: **Integers**

1.1 Lexing: breaks up the string (source code) into a list of tokens; a token is the smallest unit a parser can understand. For example, here is a list of tokens for the source code this phase can support:

1.1.1 **int** keyword
1.1.2 main identifier
1.1.3 (open parentheses
1.1.4) close parentheses
1.1.5 **{** open brace
1.1.6 **return** keyword
1.1.7 <any-integer> identifier
1.1.8 **;** semicolon
1.1.9 **}** close brace

*The lexer should accept a file and return a list of tokens.

1.2 Parsing: transforms our list of tokens into an abstract syntax tree which is one way to represent the structure of a program and the relationship between its elements. Our AST will look like this:



In addition to the AST we will need a formal grammar which defines how a series of tokens can be combined from language constructs. We will define our formal grammar using Backus-Naur Form:

```
<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <int>
```

*The parser should accept a list of tokens and return an AST, rooted at a Program node.

1.3 Code generator: this program should traverse the AST and generate the assembly code for all the elements it encounters.

	*The code generator should accept an AST and generate it assembly equivalence. 2. Phase Two: **Unary Operators** 3. Phase Three: **Binary Operators** 4. Phase Four: **Even more binary operators** ** points 2-4 will be added as soon as we deliver the first phase of the project.
Technical requirements:	1. Phase One: **Integers** 1.1 We will handle only programs with a single function **main** consisting of a single return statement. The only thing that varies is the value of the integer being returned. 1.2 The compiler will produce x86 assembly code. Thus, we won't transform the assembly into an executable file. 2. Phase Two: **Unary Operators** 3. Phase Three: **Binary Operators** 4. Phase Four: **Even more binary operators** ** points 2-4 will be added as soon as we deliver the first phase of the project.
Detailed deliveries of the project:	1. Phase One: **Integers** 1.1 Compiler: 1.1.1 Program that reads in the C file. 1.1.2 Lexer. 1.1.3 Parser. 1.1.4 Code generator. 1.1.5 Program that writes the assembly to a file. 1.2 Tests: 1.2.1 Matrix of tests 1.3 Documentation: 1.3.1 Slides documenting the construction of the compiler. 1.3.2 Project Charter document. 1.3.3 Project Scope document. 1.3.4 Analysis and Technical Review document. 1.3.5 Installation Guide document. 2. Phase Two: **Unary Operators** 3. Phase Three: **Binary Operators** 4. Phase Four: **Even more binary operators** ** points 2-4 will be added as soon as we deliver the first phase of the project.

Section III - Signatures		
Customer	Sponsor	Project Manager
Name and Signature	Name and Signature	Name and Signature