

Compiler writing in Elixir

Hernández Hernández Cristian¹, Bautista Chan Yasmin², Laparra Miranda Sandra³, Terán Hernández Aldo⁴

Abstract

Abstract: This is a first approach on the creation of a compiler in the ELIXIR programming language.

A compiler is understood as a computer program that translates a program written in a programming language into another programming language, generating an equivalent program that the machine will be able to interpret.

In this work, it will be carried out in four parts. The first consists of knowing the ELIXIR programming language in order to configure basic parts of our compiler, the architecture that we define now will facilitate some arithmetic as well as logical operations. We will work with a subproject generated by Nora Sandler on her website

Keywords: *Elixir, Compiler, help; lexing, parse, AST.*

Introduction

High-level programming languages offer many abstract programming constructs like functions, conditional statements, and loops that allow us to be incredibly productive. However, one drawback of writing code in a high-level programming language is the significant performance hit. Ideally, you should write code that is maintainable and understandable, without compromising performance. For this reason, compilers try to automatically optimize code to improve performance, and now they have a very sophisticated way of doing this. They can transform loops, conditional statements, and recursive functions, remove entire blocks of code, and take advantage of the Target Instruction Set Architecture (ISA), so code is fast and compact. It is much better to focus on writing understandable code than to perform manual optimizations that result in cryptic and difficult-to-maintain code. In fact, manually optimizing your

¹ Project Manager

² System Architect

³ System Tester

⁴ System Integrator

code might prevent the compiler from performing additional or more efficient optimizations.

Rather than manually optimizing your code, you should consider aspects of your code, such as using faster algorithms, incorporating thread-level parallelism, and using framework-specific features (for example, using move constructors).

This article is more about how to sample a simple compiler in the ELIXIR Language, the goal is not to explain how to manually optimize your code, but rather to show why you can trust the compiler to optimize your code for you.

Methods

a) Project plan

- First delivery and Second delivery.

This week the entire team will meet to analyze the specifications of the program (Project Manager, System Architect, System Integrator and System Tester), architecture, list of requirements, systems on which the program must be executed, risk analysis and how the work is going to be divided so that the objective is met of delivery on the specified date.

The team will meet to discuss the new delivery requirements and is expected to to make (minimum) a release of the advance of the compiler (commit).

- Third installment

The risk of adding the new requirements to the compiler will be evaluated (this code is on the right track) and how it would impact the system (chain errors).

- Final delivery

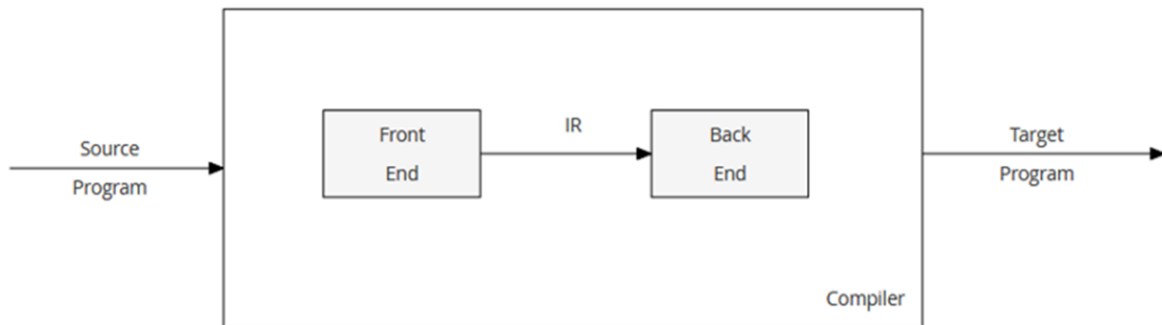
a) 22 de Diciembre 2022 (Primera entrega y segunda entrega). Versión 1.8⁵. Para esta primera entrega el compilador, la capacidad de este es limitada y solo puede leer en el "return" un entero positivo. `int main() { return ; },` además el compilador debe ser capaz de leer en el "return" constantes positivas o negativas. Soporta las siguientes Negation '-' Bitwise complement '~' Logical negation '!='

b) 2023 (Tercera entrega y cuarta entrega). Versión -- El compilador debe ser capaz de leer y hacer operaciones básicas: Addition "+" Subtraction "-" Multiplication "*" Division "/", además el compilador debe leer operaciones más complejas en el 'return': Logical AND '&&' Logical OR '||' Equal to '==' Not equal to '!=' Less than '<=' Greater than '>' Greater than or equal to '>='

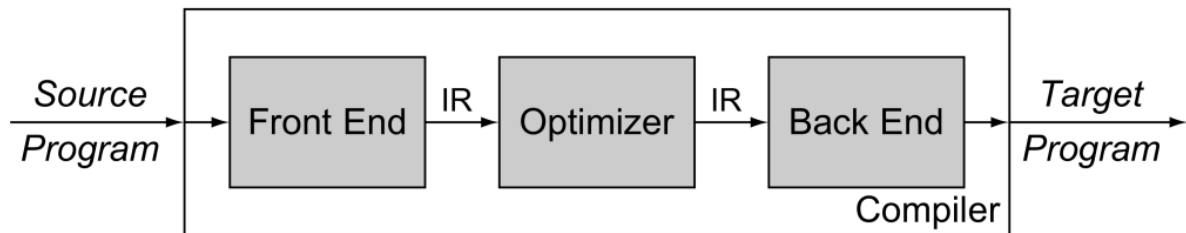
⁵ Es la ultima version que se considera en el equipo de trabajo que cumple con lo solicitado por parte de gran manager

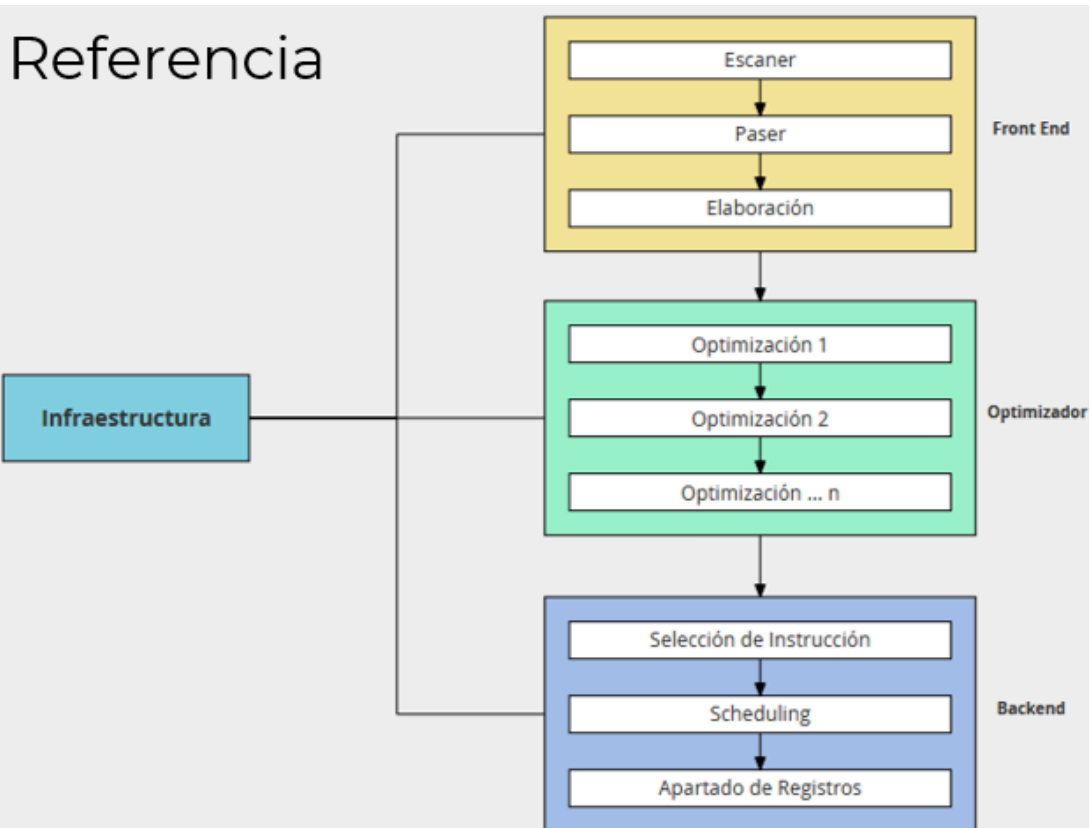
b) Compiler architecture

The main goal of our compiler is to preserve the meaning of the input of the program and in some cases most of the time, we must improve the output of the program to make processes more efficient and with this, to better use the processes of the computer.



The infrastructure of our compiler is based on a base of 3 structures: Font end, Optimizer and Back end as seen in the diagram in a linear manner as shown indicate in the arrows.





The compiler implementation is based on traditional compiler development, that is that is, lexical analysis (on its output it returns a list of tokens), syntactic analysis (parses the statements using an AST tree) and finally code generation (which generates the assembler of said input) which at its output will generate an executable file.



c) Test plan and test suites

(Configuración, compilación y ejecución)

• Prerrequisitos

Tener sistema operativo Unix

Instalar Git

Instalar Elixir

- Instalación de Git

Para poder instalar Git podemos acceder a la página oficial de Git y encontrar las instrucciones para el sistema operativo Unix que tengamos.

<https://git-scm.com/download/linux>

- Instalación Elixir

De igual forma podemos instalar Elixir a través de la página oficial:

<https://elixir-lang.org/install.html#unix-and-unix-like>

En esta página encontraremos la línea que debemos correr para que se nos instale Elixir.

- ❖ Compilacion y Ejecucion

Para obtener los archivos del compilador necesitamos primero clonarlos del repositorio de git, para esto abriremos la terminal y nos posicionaremos en la dirección en la que queremos que se clone el repositorio y pondremos la

1. Entry of file with extension “.c” with content and its output must be an executable.

```
int main() {  
    return <entero>;  
}
```

2. It must be run from the command line or terminal for unix-like systems.

3. The executable must have the same name as the compiled file, it must be in the same path that was compiled.

4. With the flag “-o <executable_name>” it must be possible to modify the name of the executable by a different one than the original file. The name of the executable by default is the name of the original file.

```
$ ./nqcc -o <nombre_ejecutable>
```

5. The “-s” flag returns the assembler of the compiled file and does not generate a executable.

```
$ ./nqcc -s <nombre_archivo>
```

6. The “-t” flag returns the list of tokens graphically on the screen, it does not generate file nor executable of the compiled file.

```
$ ./nqcc --o <nombre_ejecutable>
```

7. The “-a” flag returns the AST tree of the parser in console and is not generated executable of the compiled file.

```
$ ./nqcc --a <nombre_archivo>
```

8. The “-help” flag returns compiler instructions.

```
$ ./nqcc --help
```

9. If you type “\$ compiler ” in the console and press enter without specifying the name of the file to compile, marks an error and returns the message "missing name of the File, Archive".

```
$ ./nqcc  
$ Falta nombre del archivo
```

10. If “\$ compiler <file name>” is typed in the console, it returns the executable of the compiled file in the same folder as the .c file.

11. Once compiled if there are errors in the file or reserved words were inserted wrong, a semicolon is missing, the main braces were not closed, or there are other words that should not go inside the file, it shows a message “lexical error or syntactic” indicating the line number in the console.

```
$ ./nqcc <nombre_archivo>  
$ Error error léxico o sintáctico, línea --
```

12. The command line only supports 1 flag at a time and no more can be chained flags to the instruction.

13. A manual will be delivered to run the tests and also another manual on how to compile the program (such as cloning and running the tests from the git examples of Nora Sandler).

14. Each delivery must have its version (git tag version) for the client to consult and thus know which version is the one you are compiling.

Result and discusión

```
1  int main()  
2  {  
3      return ~!4;  
4  }
```

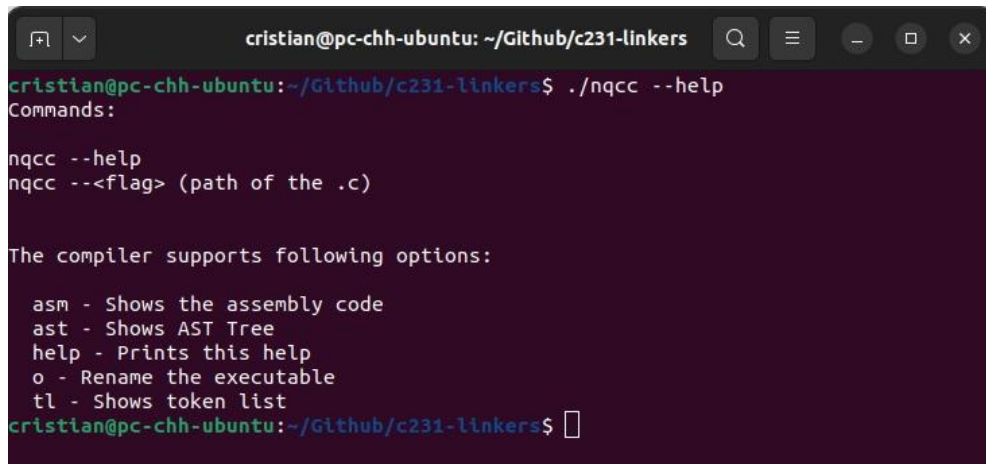
a) Lista de tokens

```
cristian@pc-chh-ubuntu: ~/Github/c231-linkers  
cristian@pc-chh-ubuntu:~/Github/c231-linkers$ ./nqcc --tl ./examples/return_2.c  
Token List: ./examples/return_2.c  
  
Lexer output (Token List): [  
  {:int_keyword, 1},  
  {:main_keyword, 1},  
  {:open_paren, 1},  
  {:close_paren, 1},  
  {:open_brace, 2},  
  {:return_keyword, 3},  
  {:negation, 3},  
  {:bitwise_complement, 3},  
  {:logical_negation, 3},  
  {:constant, 4},  
  {:semicolon, 3},  
  {:close_brace, 4}  
]  
cristian@pc-chh-ubuntu:~/Github/c231-linkers$
```

b) Árbol AST

```
cristian@pc-chh-ubuntu: ~/Github/c231-linkers  
Tree AST: ./examples/return_2.c  
Parser output (Tree AST): NAST(  
  left_node: NAST(  
    left_node: NAST(  
      left_node: NAST(  
        left_node: NAST(  
          left_node: nil,  
          node_name: :constant,  
          right_node: nil,  
          value: 4  
        ),  
        node_name: :logical_negation,  
        right_node: nil,  
        value: nil  
      ),  
      node_name: :bitwise_complement,  
      right_node: nil,  
      value: nil  
    ),  
    node_name: :negation,  
    right_node: nil,  
    value: nil  
  ),  
  node_name: :return,  
  right_node: nil,  
  value: nil  
,  
  node_name: :function,  
  right_node: nil,  
  value: :main  
,  
  node_name: :program,  
  right_node: nil,  
  value: nil  
)
```

c) Ayuda



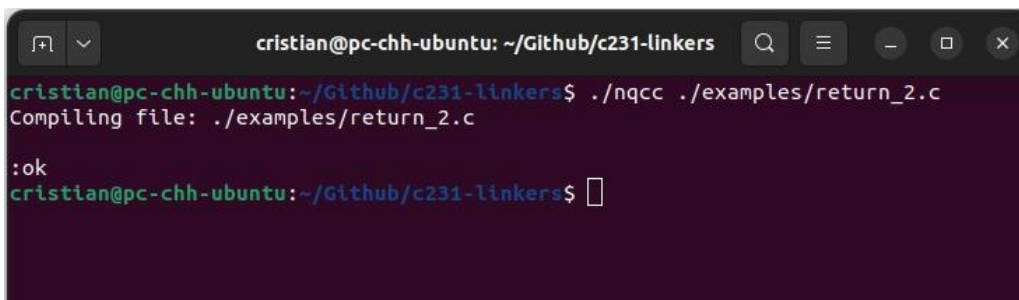
```
cristian@pc-chh-ubuntu: ~/Github/c231-linkers
cristian@pc-chh-ubuntu:~/Github/c231-linkers$ ./nqcc --help
Commands:

nqcc --help
nqcc --<flag> (path of the .c)

The compiler supports following options:

asm - Shows the assembly code
ast - Shows AST Tree
help - Prints this help
o - Rename the executable
tl - Shows token list
cristian@pc-chh-ubuntu:~/Github/c231-linkers$
```

d) Ejecución correcta



```
cristian@pc-chh-ubuntu: ~/Github/c231-linkers
cristian@pc-chh-ubuntu:~/Github/c231-linkers$ ./nqcc ./examples/return_2.c
Compiling file: ./examples/return_2.c

:ok
cristian@pc-chh-ubuntu:~/Github/c231-linkers$
```

e) Error Léxico



```
cristian@pc-chh-ubuntu: ~/Github/c231-linkers
cristian@pc-chh-ubuntu:~/Github/c231-linkers$ ./nqcc ./examples/return_2.c
Compiling file: ./examples/return_2.c

"Error lexico: (in) se encuentra un token no reconocido, en línea: 1"
"Error lexico: (retur) se encuentra un token no reconocido, en línea: 3"
cristian@pc-chh-ubuntu:~/Github/c231-linkers$
```

f) En el caso de un error sintáctico, el mensaje muestra entre paréntesis la línea donde se encuentra el error, mientras que el mensaje descriptivo dependerá del tipo de falla detectado. Finalmente el "Cerca de: --" indica el token más cercano en donde esta el error.


```
cristian@pc-chh-ubuntu: ~/Github/c231-linkers
cristian@pc-chh-ubuntu:~/Github/c231-linkers$ ./nqcc ./examples/return_2.c
Compiling file: ./examples/return_2.c

ERROR SINTACTICO
"En linea (3), Error: falta la palabra clave 'return'. Cerca de: negation"
cristian@pc-chh-ubuntu:~/Github/c231-linkers$
```

"En linea (num línea), Error: mensaje descriptivo. Cerca de: token cercano"

g) Código ensamblador

```
cristian@pc-chh-ubuntu: ~/Github/c231-linkers
cristian@pc-chh-ubuntu:~/Github/c231-linkers$ ./nqcc --asm ./examples/return_2.c
Assembler Code: ./examples/return_2.c

Code generator output (Assembler Code):

.section .text.startup,"ax",@progbits
.p2align 4
.globl main
main:
    endbr64
    movl $4, %eax
    cmpl $0, %eax
    movl $0, %eax
    sete %al
    not %eax
    neg %eax
    ret
.section .note.GNU-stack,"",@progbits

cristian@pc-chh-ubuntu:~/Github/c231-linkers$
```

Conclusión

References

Nora Sandler. (2017, 29 noviembre). <https://norasandler.com/2017/11/29/Write-aCompiler.html>