How to create a compiler by hexacode

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

INTRODUCTION

FOR OUR COMPILER CLASS, WE MUST DEVELOP A COMPILER, DURING THE SEMESTER AND WE HAVE TO ACCOMPLISH WITH FOUR DELIVERIES, IN SPECIFIC, IN THIS PRESENTATION WE ARE GOING TO PRESENT THE FIRST DELIVERY, FOR THE DEVELOP OF THIS PROJECT, WE NEEDED TO DIVIDE THE TEAM IN SPECIFIC FUNCTIONS FOR EACH OF THE MEMBERS OF THE TEAM.

OUR TEAM IS COMPOSED OF:

-PROJECT MANAGER

-SYSTEM ARCHITECT

-SYSTEM INTEGRATOR

-TESTER

-DEVELOPERS

FOR THE EVALUATION OF THIS PROJECT, WE HAVE A CLIENT, WHO WILL BE IN CHARGE OF EVALUATE, VERIFY AND CHECK THAT EVERY DETAIL REQUESTED FOR THIS COMPILER PROJECT IS IN ORDER AND CORRECT.

DEVELOPERS

Fernando - Project Manager -

Niver - Architect - Code Generator - Sanitizer - Helping with Lexer Test

Martínez Vázquez Diego - System Integrator - Parser, Parser Test, helping with Linker

Víctor - Tester - Helping with Parser - Helping with Code Generator

Díaz Guerrero Alan Mauricio - Developer - Lexer

PROJECT OVERVIEW -FIRST DELIVERY

For this first delivery, our goal is to accomplish with a compiler that can validate any integer.

The compiler it should be able of read the code and generate an executable.

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

PROJECT PLAN

Plan organization

Primera entrega

Proyecto de Compiladores

HORA DE INICIO 9:00

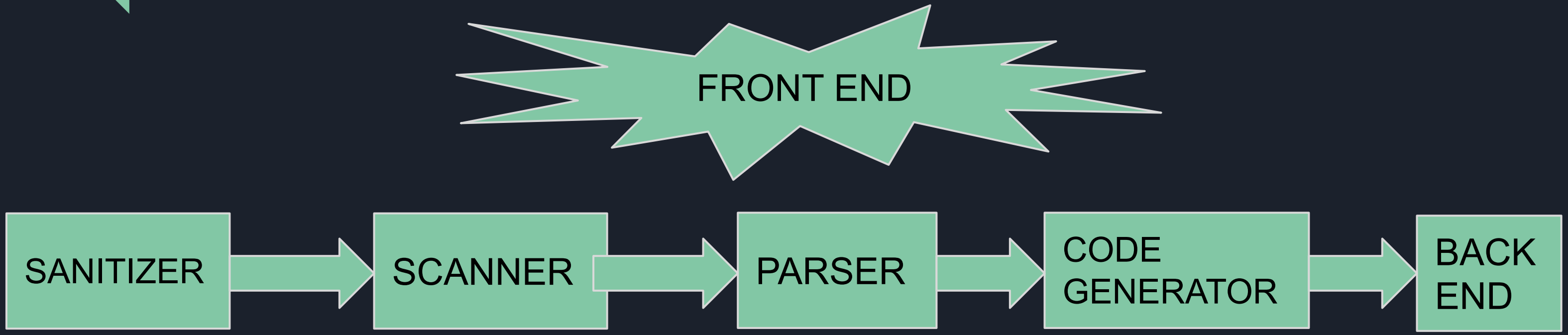
INTERVALO DE TIEMPO 60 (en minutos)

2022

HORA	LUN	MAR	MIÉ	JUE	VIE	SÁB	DOM
21:00-23:30							Trabajo en Equipo sesion Zoom
19:00-21:00		Trabajo en Equipo sesion Zoom					
16:00-18:00						Trabajo en Equipo sesion Zoom	

COMPILER ARCHITECTURE

Following the steps of how to make a compiler, we designed our compiler based on these steps.



COMPILER ARCHITECTURE

Compiler Composition

COMPILER STRUCTURE

For a correct development of our Parser, we had to follow in a correct manner the production rules.

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

COMPILER STRUCTURE

Sanitizer: The main idea of this program is to clear the tabs and the empty spaces from the file content.

```
defmodule Sanitizer do

  def sanitize_source(file_content) do
    trimmed_content = String.trim(file_content)
    Regex.split(~r/\s+/, trimmed_content)
  end
end
```

COMPILER STRUCTURE -LEXER

The main objective is to receive the file and converts it in a token list, that verifies that it exists in the grammar language.

```
defmodule Lexer do

  def scan_words(words) do
    Enum.flat_map(words, &lex_raw_tokens/1)
  end


  def get_constant(program) do
    case Regex.run(~r/^\d+/, program) do
      [value] ->
        {{:constant, String.to_integer(value)}, String.trim_leading(program, value)}

      program ->
        {:error, "Token not valid: #{program}"}
    end
  end
```

```elixir
def lex_raw_tokens(program) when program != "" do
  {token, rest} =
    case program do
      "{" <> rest ->
        {:open_brace, rest}

      "}" <> rest ->
        {:close_brace, rest}

      "(" <> rest ->
        {:open_paren, rest}

      ")" <> rest ->
        {:close_paren, rest}

      ";" <> rest ->
        {:semicolon, rest}

      "return" <> rest ->
        {:return_keyword, rest}

      "int" <> rest ->
        {:int_keyword, rest}

      "main" <> rest ->
        {:main_keyword, rest}
```

```elixir
        rest ->
          get_constant(rest)
      end

    if token != :error do
      remaining_tokens = lex_raw_tokens(rest)
      [token | remaining_tokens]
    else
      [:error]
    end
  end

  def lex_raw_tokens(_program) do
    []
  end
end
```

COMPILER STRUCTURE - PARSER

The main objective of the parser is to verify that the order of the token list is correct, that any part of it is missing and it provides the AST structure to the code generator.

```
defmodule Parser do

  def parse_program(token_list) do
    function = parse_function(token_list)

    case function do
      {{:error, error_message}, _rest} ->
        {:error, error_message}

      {function_node, rest} ->
        if rest == [] do
          %AST{node_name: :program, left_node: function_node}
        else
          {:error, "Error: there are more elements after function end"}
        end
    end
  end
```

```elixir
def parse_function([next_token | rest]) do
  if next_token == :int_keyword do
    [next_token | rest] = rest
    if next_token == :main_keyword do
      [next_token | rest] = rest
      if next_token == :open_paren do
        [next_token | rest] = rest
        if next_token == :close_paren do
          [next_token | rest] = rest
          if next_token == :open_brace do
            statement = parse_statement(rest)

            case statement do
              {{:error, error_message}, rest} ->
                {{:error, error_message}, rest}

              {statement_node, [next_token | rest]} ->
                if next_token == :close_brace do
                  {%AST{node_name: :function, value: :main, left_node: statement_node}, rest}
                else
                  {{:error, "Error, close brace missed"}, rest}
                end
              _ -> {{:error, "Error, close brace missed"}, rest}
            end
          else
            {{:error, "Error: open brace missed"}, rest}
```

COMPILER STRUCTURE - PARSER

```
              end
            else
              {{:error, "Error: close parentesis missed"}, rest}
            end
          else
            {{:error, "Error: open parentesis missed"}, rest}
          end
        else
          {{:error, "Error: main function missed"}, rest}
        end
      else
        {{:error, "Error, return type value missed"}, rest}
      end
    end
```

COMPILER STRUCTURE - PARSER

```
def parse_statement([next_token | rest]) do
  if next_token == :return_keyword do
    expression = parse_expression(rest)

    case expression do
      {{:error, error_message}, rest} ->
        {{:error, error_message}, rest}

      {exp_node, [next_token | rest]} ->
        if next_token == :semicolon do
          {%AST{node_name: :return, left_node: exp_node}, rest}
        else
          {{:error, "Error: semicolon missed after constant to finish return statement"}, rest}
        end
    end
  else
    {{:error, "Error: return keyword missed"}, rest}
  end
end
```

COMPILER STRUCTURE - PARSER

```
  def parse_expression([next_token | rest]) do
    case next_token do
      {:constant, value} -> {%AST{node_name: :constant, value: value}, rest}
      _ -> {{:error, "Error: constant value missed"}, rest}
    end
  end
end
```

COMPILER STRUCTURE - CODE GENERATOR

```elixir
defmodule CodeGenerator do
@spec generate_code(atom | %{:left_node => atom | %{:left_node => atom | %{:left_node => atom | map, :node_name => :constant | :function | :program | :ret
    def generate_code(ast) do
        {_,esp}=:os.type()
        post_order(ast,esp)
    end

@spec post_order(atom | %{:left_node => atom | %{:left_node => atom | %{:left_node => atom | map, :node_name => :constant | :function | :program | :return,
    def post_order(node,esp) do
        case node do
            nil ->
                nil
            ast_node ->
              code_snippet = post_order(ast_node.left_node,esp)
                emit_code(ast_node.node_name,code_snippet ,ast_node.value,esp)
        end
    end
    @spec emit_code(:constant | :function | :program | :return, any, any, any) :: any
    def emit_code(:program, code_snippet , _, os) when os == :darwin do
        """

            .section        __TEXT,__text,regular,pure_instructions
            .p2align        4, 0x90
        """ <>
        code_snippet
    end
```

COMPILER STRUCTURE - CODE GENERATOR

The main objective of the code generator is of depending on the operative system and computer processor, the code generator is going to receive the ast tree and create an assembly code for each architecture.

```
def emit_code(:program, code_snippet , _, _) do

    code_snippet
end

def emit_code(:function, code_snippet , :main, os) when os == :linux do
    """
        .globl  main
    main:
    """ <>
    code_snippet
end

def emit_code(:function, code_snippet , :main, _) do
    """
        .globl  _main
    _main:
    """ <>
    code_snippet
end
def emit_code(:return, code_snippet, _, _) do
    """
        movl    #{code_snippet }, %eax
        ret
    """
end
```

COMPILER STRUCTURE - CODE GENERATOR

```
  def emit_code(:constant, _, value, _) do
    "$#{value}"
  end

end
```

COMPILER STRUCTURE - LINKER

The linker is used for work with the gcc for create the executable.

```
defmodule Linker do

@moduledoc """
  Documentation for Linker.
"""


  def generate_binary(assembler, assembly_path) do
    assembly_file_name = Path.basename(assembly_path)
    binary_file_name = Path.basename(assembly_path, ".s")
    output_dir_name = Path.dirname(assembly_path)
    assembly_path = output_dir_name <> "/" <> assembly_file_name

    File.write!(assembly_path, assembler)
    System.cmd("gcc", [assembly_file_name, "-o#{binary_file_name}"], cd: output_dir_name)
    File.rm!(assembly_path)
  end
end
```

COMPILER STRUCTURE - NQCC

```elixir
defmodule Nqcc do
  @moduledoc """
  Documentation for Nqcc.
  """
  @commands %{
    "help" => "Prints this help"
  }

  def main(args) do
    args
    |> parse_args
    |> process_args
  end


  def parse_args(args) do
    OptionParser.parse(args, switches: [help: :boolean])
  end

  defp process_args({[help: true], _, _}) do
    print_help_message()
  end

  defp process_args({_, [file_name], _}) do
    compile_file(file_name)
  end
```

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.

COMPILER STRUCTURE - NQCC

```
defp compile_file(file_path) do
  IO.puts("Compiling file: " <> file_path)
  assembly_path = String.replace_trailing(file_path, ".c", ".s")

  File.read!(file_path)
  |> Sanitizer.sanitize_source()
  |> IO.inspect(label: "\nSanitizer ouput")
  |> Lexer.scan_words()
  |> IO.inspect(label: "\nLexer ouput")
  |> Parser.parse_program()
  |> IO.inspect(label: "\nParser ouput")
  |> CodeGenerator.generate_code()
  |> Linker.generate_binary(assembly_path)
end

defp print_help_message do
  IO.puts("\nnqcc --help file_name \n")

  IO.puts("\nThe compiler supports following options:\n")

  @commands
  |> Enum.map(fn {command, description} -> IO.puts("  #{command} - #{description}") end)
end
end
```

TESTS AND RESULTS

With the help of the test programs included in our project we can verify the function of every module, with this we can verify the behaviour of our program for future modifications.

TEST -LEXER

```elixir
defmodule LexerTest do
  use ExUnit.Case
  doctest Lexer

  setup_all do
    {:ok,
     tokens: [
       :int_keyword,
       :main_keyword,
       :open_paren,
       :close_paren,
       :open_brace,
       :return_keyword,
       {:constant, 2},
       :semicolon,
       :close_brace
     ]}
  end
```

```elixir
# tests to pass
test "return 2", state do
  code = """
    int main() {
      return 2;
  }
  """

  s_code = Sanitizer.sanitize_source(code)

  assert Lexer.scan_words(s_code) == state[:tokens]
end
```

TEST -LEXER

```
test "return 0", state do
  code = """
  int main() {
    return 0;
  }
  """

  s_code = Sanitizer.sanitize_source(code)

  expected_result = List.update_at(state[:tokens], 6, fn _ -> {:constant, 0} end)
  assert Lexer.scan_words(s_code) == expected_result
end
```

TEST -LEXER

```
test "multi_digit", state do
  code = """
  int main() {
    return 100;
  }
  """

  s_code = Sanitizer.sanitize_source(code)

  expected_result = List.update_at(state[:tokens], 6, fn _ -> {:constant, 100} end)
  assert Lexer.scan_words(s_code) == expected_result
end
```

TEST -LEXER

```
test "new_lines", state do
  code = """
  int
  main
  (
  )
  {
  return
  2
  ;
  }
  """


  s_code = Sanitizer.sanitize_source(code)

  assert Lexer.scan_words(s_code) == state[:tokens]
end
```

TEST -LEXER

```
test "no_newlines", state do
  code = """
  int main(){return 2;}
  """

  s_code = Sanitizer.sanitize_source(code)

  assert Lexer.scan_words(s_code) == state[:tokens]
end

test "spaces", state do
  code = """
  int   main    (  )  {   return  2 ; }
  """

  s_code = Sanitizer.sanitize_source(code)

  assert Lexer.scan_words(s_code) == state[:tokens]
end
```

TEST -LEXER

```
test "elements separated just by spaces", state do
  assert Lexer.scan_words(["int", "main(){return", "2;}"]) == state[:tokens]
end

test "function name separated of function body", state do
  assert Lexer.scan_words(["int", "main()", "{return", "2;}"]) == state[:tokens]
end

test "everything is separated", state do
  assert Lexer.scan_words(["int", "main", "(", ")", "{", "return", "2", ";", "}"]) ==
           state[:tokens]
end
```

TEST -LEXER

```
# tests to fail
test "wrong case", state do
  code = """
  int main() {
    RETURN 2;
  }
  """

  s_code = Sanitizer.sanitize_source(code)

  expected_result = List.update_at(state[:tokens], 5, fn _ -> :error end)
  assert Lexer.scan_words(s_code) == expected_result
end
end
```

TEST - PARSER

```elixir
defmodule ParserTest do
  use ExUnit.Case
  doctest Parser

  setup_all do
    {:ok,
    ast:  %AST{
      left_node: %AST{
        left_node: %AST{
          left_node: %AST{
            left_node: nil,
            node_name: :constant,
            right_node: nil,
            value: 2
          },
          node_name: :return,
          right_node: nil,
          value: nil
        },
        node_name: :function,
        right_node: nil,
        value: :main
      },
      node_name: :program,
      right_node: nil,
      value: nil
```

```elixir
    },
    ast1:  %AST{
      left_node: %AST{
        left_node: %AST{
          left_node: %AST{
            left_node: nil,
            node_name: :constant,
            right_node: nil,
            value: 0
          },
          node_name: :return,
          right_node: nil,
          value: nil
        },
        node_name: :function,
        right_node: nil,
        value: :main
      },
      node_name: :program,
      right_node: nil,
      value: nil
    },
```

```elixir
    },
    ast2:  %AST{
      left_node: %AST{
        left_node: %AST{
          left_node: %AST{
            left_node: nil,
            node_name: :constant,
            right_node: nil,
            value: 100
          },
          node_name: :return,
          right_node: nil,
          value: nil
        },
        node_name: :function,
        right_node: nil,
        value: :main
      },
      node_name: :program,
      right_node: nil,
      value: nil
    },
```

TEST - PARSER

```
    },
    error_no_int: {:error, "Error, return type value missed"},
    error_no_main: {:error, "Error: main function missed"},
    error_no_open_paren: {:error, "Error: open parentesis missed"},
    error_no_close_paren: {:error, "Error: close parentesis missed"},
    error_no_open_brace: {:error, "Error: open brace missed"},
    error_no_return: {:error, "Error: return keyword missed"},
    error_no_number: {:error, "Error: constant value missed"},
    error_no_semicolon: {:error, "Error: semicolon missed after constant to finish return statement"},
    error_no_close_brace: {:error, "Error, close brace missed"},
    error_more_elements: {:error, "Error: there are more elements after function end"},
  }
end
```

TEST - PARSER

```elixir
# tests to pass
test "return 2", state do
  token_list = [
    :int_keyword,
    :main_keyword,
    :open_paren,
    :close_paren,
    :open_brace,
    :return_keyword,
    {:constant, 2},
    :semicolon,
    :close_brace
  ]

  assert Parser.parse_program(token_list) == state[:ast]
end
```

TEST - PARSER

```
test "return 0", state do
  token_list = [
    :int_keyword,
    :main_keyword,
    :open_paren,
    :close_paren,
    :open_brace,
    :return_keyword,
    {:constant, 0},
    :semicolon,
    :close_brace
  ]

  assert Parser.parse_program(token_list) == state[:ast1]
end
```

TEST - PARSER

```elixir
test "multi_digit", state do
  token_list = [
    :int_keyword,
    :main_keyword,
    :open_paren,
    :close_paren,
    :open_brace,
    :return_keyword,
    {:constant, 100},
    :semicolon,
    :close_brace
  ]

  assert Parser.parse_program(token_list) == state[:ast2]
end
```

TEST - PARSER

```
test "new_lines", state do
  token_list = [
    :int_keyword,
    :main_keyword,
    :open_paren,
    :close_paren,
    :open_brace,
    :return_keyword,
    {:constant, 2},
    :semicolon,
    :close_brace
  ]
  assert Parser.parse_program(token_list) == state[:ast]
end
```

TEST - PARSER

```
test "no_newlines", state do
  token_list = [
    :int_keyword,
    :main_keyword,
    :open_paren,
    :close_paren,
    :open_brace,
    :return_keyword,
    {:constant, 2},
    :semicolon,
    :close_brace
  ]
  assert Parser.parse_program(token_list) == state[:ast]
end
```

TEST - PARSER

```
# tests to fail
test "no_int_keyword", state do
  token_list=[
      :main_keyword,
      :open_paren,
      :close_paren,
      :open_brace,
      :return_keyword,
      {:constant, 2},
      :semicolon,
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_no_int]
end
```

TEST - PARSER

```
test "no_main", state do
  token_list=[
      :int_keyword,
      :open_paren,
      :close_paren,
      :open_brace,
      :return_keyword,
      {:constant, 2},
      :semicolon,
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_no_main]
end
```

TEST - PARSER

```
test "missing_close_paren", state do
  token_list=[
      :int_keyword,
      :main_keyword,
      :open_paren,
      :open_brace,
      :return_keyword,
      {:constant, 2},
      :semicolon,
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_no_close_paren]
end
```

TEST - PARSER

```
test "no_open_brace", state do
  token_list=[
      :int_keyword,
      :main_keyword,
      :open_paren,
      :close_paren,
      :return_keyword,
      {:constant, 2},
      :semicolon,
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_no_open_brace]
end
```

TEST - PARSER

```
test "no_return", state do
  token_list=[
      :int_keyword,
      :main_keyword,
      :open_paren,
      :close_paren,
      :open_brace,
      {:constant, 2},
      :semicolon,
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_no_return]
end
```

TEST - PARSER

```
test "no_number", state do
  token_list=[
      :int_keyword,
      :main_keyword,
      :open_paren,
      :close_paren,
      :open_brace,
      :return_keyword,
      :semicolon,
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_no_number]
end
```

TEST - PARSER

```
test "no_semicolon", state do
  token_list=[
      :int_keyword,
      :main_keyword,
      :open_paren,
      :close_paren,
      :open_brace,
      :return_keyword,
      {:constant, 2},
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_no_semicolon]
end
```

TEST - PARSER

```
test "no_close_brace", state do
  token_list=[
      :int_keyword,
      :main_keyword,
      :open_paren,
      :close_paren,
      :open_brace,
      :return_keyword,
      {:constant, 2},
      :semicolon]
      assert Parser.parse_program(token_list) == state[:error_no_close_brace]
end
```

TEST - PARSER

```
test "more element", state do
  token_list=[
      :int_keyword,
      :main_keyword,
      :open_paren,
      :close_paren,
      :open_brace,
      :return_keyword,
      {:constant, 2},
      :semicolon,
      :close_brace,
      :close_brace]
      assert Parser.parse_program(token_list) == state[:error_more_elements]
end
end
```

CONCLUSIONS

Martínez Vázquez DIego: At first it was tough to develop this project because I had no to much knowledge about compilers, and its structure, we had to start to look for information on the internet to start to understand how to use Github and Elixir language, when our teacher Norberto showed us an example of the compiler for the first delivery, we could start to understand more clear what exactly we had to deliver, we understood the main components for our compiler with sanitizer, lexer, parser, code generator and linker. When we understood that thing, we were ready to start to develop our own compiler using the base compiler of our teacher. When I started to develop my parts of the compiler, I was understanding that it is really interesting how each detail counts, like all the empty spaces, or a missing token, something really interesting is the develop of the parser, because we have to be very careful with any missing token, I think that is the most interesting I learnt from it.

CONCLUSIONS

Ceballos Ricardo Fernando:

This project was a great challenge, since at the beginning I did not know a possible solution for the implementation, but from an arduous investigation, I understood the different modules that make up a compiler, after knowing the operation of each one of its components, I proceeded to investigate the operation of elixir and together with the knowledge of the professor we managed to develop this basic compiler for language c.

CONCLUSIONS

Díaz Guerrero Alan Mauricio: While developing this project we had to endure and face lots of different kinds of problems which delayed the delivery of it. We had to comprehend most of the code we were working with, and learn all of the terminology like the modules or sintaxis of Elixir. Which is a language that none of the team members were familiarized with, so we had to organize studying sessions to be at the same level of knowledge regarding the program.

We tried to understand how each module worked, and how it works we the others to have a complete perception of all the program. As planned we got a full view of it, and could get it to compile the C++ code.

CONCLUSIONS

Niver Asaid Martínez Hernández: We look forward to being able to develop the remaining phases with much more success, build a project that convinces the client and leaves us a great deal of learning, and to be able to do a consistent review of the implementation we are doing.

From another point of view, also to achieve satisfaction with the new learnings and results consistent with good development practices in this implementation of a compiler.