First Delivery

Barrientos Veana Luis Mauricio.

González Pacheco Leonardo Alonso.

Martínez Matías Joan Eduardo.

Rosales Romero Ricardo.

Objective:

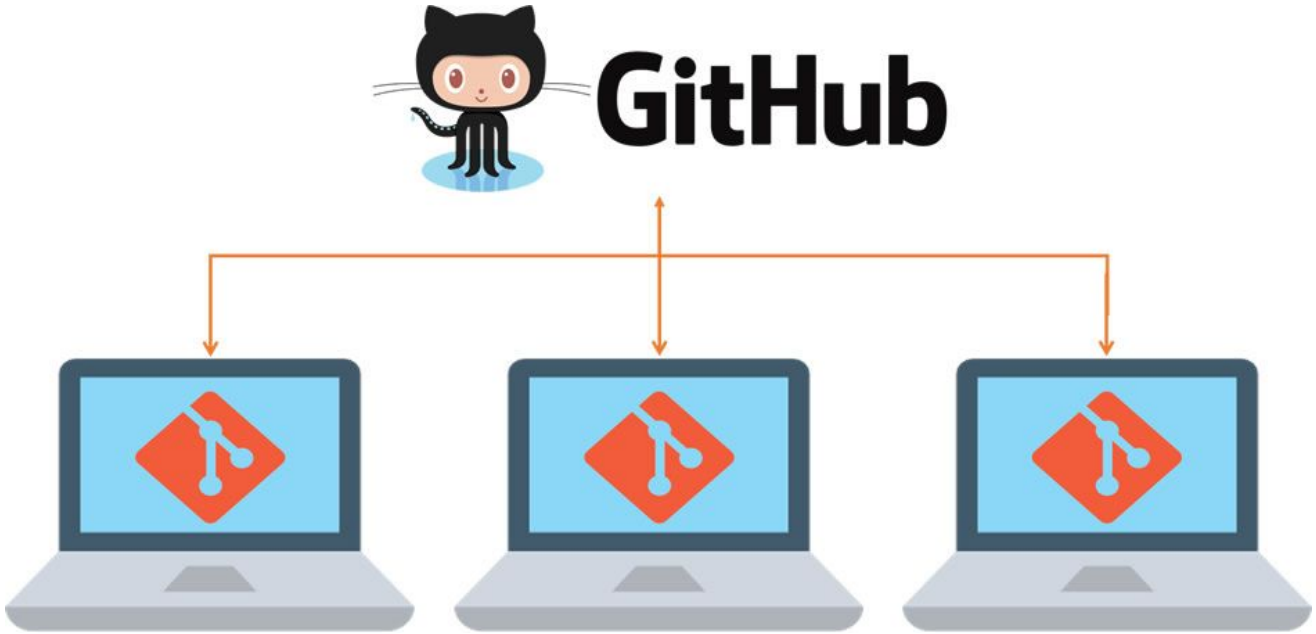
Develop a C compiler that meets the requirements of the client Norberto Jesús Ortigoza Márquez; the project will be developed in elixir.

The delivery dates below will be provided in the work plan.

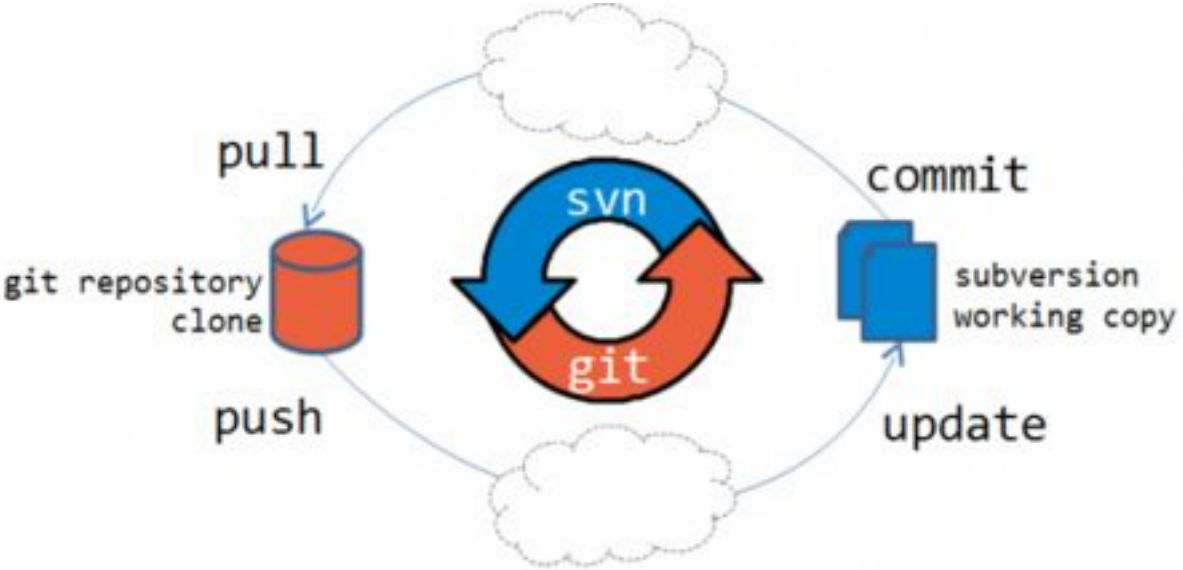
Gremlins Working P8lan

# Activity	Activity	February / 3-7 / 2020 (1)	February / 10-14 / 2020 (2)	February / 17-21 / 2020 (3)	February / 24-28 / 2020 (4)	March / 2-6 / 2020 (5)	March / 9-13 / 2020 (6)	March / 16-20 / 2020 (7)	23-27 / marzo / 2020 (8)	February / 30 - April / 3 / 2020 (9)	April / 6-10 / 2020 (10)	April / 13-17 / 2020 (11)	April / 20-24 / 2020 (12)	April / 27 - May / 1 / 2020 (13)	May / 4-8 / 2020 (14)	May / 11-15 / 2020 (15)	May / 18-22 / 2020 (16)	Resourses
1	Read Elixir Manual	Tue	Thu															All
2	Requirements	Tue		Tue	Tue													Project Manayer
3	Validate the Requirements	Tue																All
4	Analysis		Mon	Thu	Thu													All
5	Architecture Design		Thu		Fri													System Architect
6	Lexico Analysis			Sun		Sun												All
7	Global Funtion (Main)																	All
8	Lifting of Requirements					Mon												Project Manayer
9	First Delivery						Tue	Thu										All

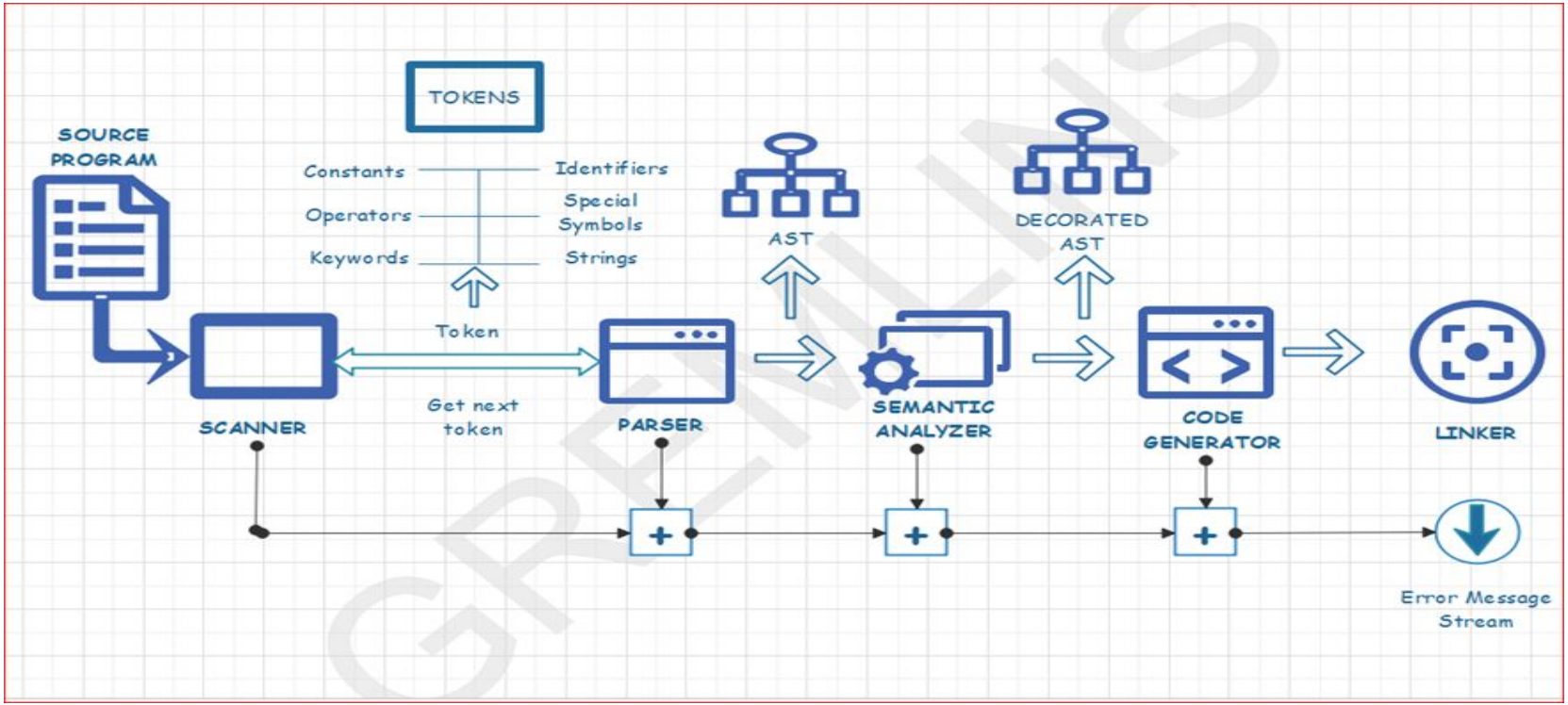
Use of git and github



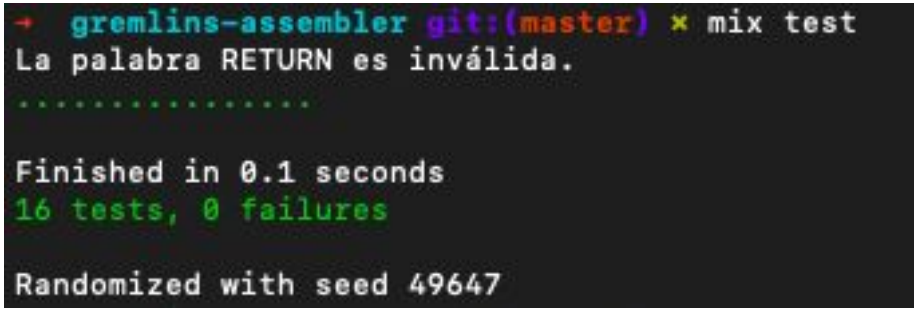
Use of git and github



Architecture (Pipe-filter pattern)



Tests


→ gremlins-assembler git:(master) × mix test
La palabra RETURN es inválida.
.....
Finished in 0.1 seconds
16 tests, 0 failures
Randomized with seed 49647

Complications

There were problems with github since the invitations provided by the teacher were not initially accepted.

The planning in the dates of deliveries and meetings was complicated by the times of each member.

To carry out the code, the functionality of all the parts of the compilation had to be correctly understood.

Second Delivery

Barrientos Veana Luis Mauricio.

González Pacheco Leonardo Alonso.

Martínez Matías Joan Eduardo.

Rosales Romero Ricardo.

Aspects that were improved.

Organization.

Handling tools.

Planning for more realistic and achievable goals.

Changes

3 unary operators added.

Negation
Complementary bit
Logical negation

Activities.

Code update.

Changes in git and Github.

Documentation was improved and the objectives of the second installment were explained more clearly.

Coupling Architecture

```
def manager(file, path, opt) do
#Utilizando "with" se procesa el archivo. Si hay error deja de hacer la compilación.
with  {:ok, tok} <- Lexer.scan_word(file, opt),
      {:ok , ast} <- Parser.parse_token_list(tok, opt),
      {:ok, asm} <- Generador.code_gen(ast, opt, path),
      {:ok, _}  <- Linker.outputBin(asm, opt, path)
      do
      IO.puts("Finalizó la compilación de forma exitosa.")
else
#Se muestra el motivo del error o la salida de la opción seleccionada al compilar
      {:error, error} -> IO.puts(error)
      {:only_tokens, _} -> IO.puts("Lista de tokens.")
      {:only_ast, _} -> IO.puts("Árbol Sintáctico.")
      {:only_asm, path_asm} ->IO.puts(path_asm)
```

Adding Unary Operators to Lexer

```
def lex_raw_tokens(program) when program != "" do #Búsq
  {token, resto} =
  case program do
      "{" <> resto -> {:open_brace, resto}
      "}" <> resto -> {:close_brace, resto}
      "(" <> resto -> {:open_par, resto}
      ")" <> resto-> {:close_par, resto}
      ";" <> resto -> {:semicolon, resto}
      "return" <> resto -> {:return_Reserveword, resto}
      "int" <> resto -> {:int_Reserveword, resto}
      "main" <> resto -> {:main_Reserveword, resto}
      "-" <> resto -> {:negation_Reserveword, resto}
      "!" <> resto -> {:logicalNeg, resto}
      "~" <> resto -> {:bitewise_Reserveword, resto}
```

Adding Unary Operators to Parser

We created a new function , that benefits the controll and order to the process for creating nodes :

```
def pars_factor(tokens) do
  #Parseando con operador unario
  if List.first(tokens) == :negation_Reserveword or List.first(tokens) == :bitewise_Reserveword  or List.first(tokens) == :logicalNeg
```

Adding Unary Operators to Code Generator

```
def codigo_gen(:negation_Reserveword, _, codigo, _) do
  codigo <> """
    neg  %eax
  """
end

def codigo_gen(:logicalNeg, _, codigo, _) do
  codigo <> """
    cmp     $0, %rax
    mov     $0, %rax
    sete    %al
  """
end
```

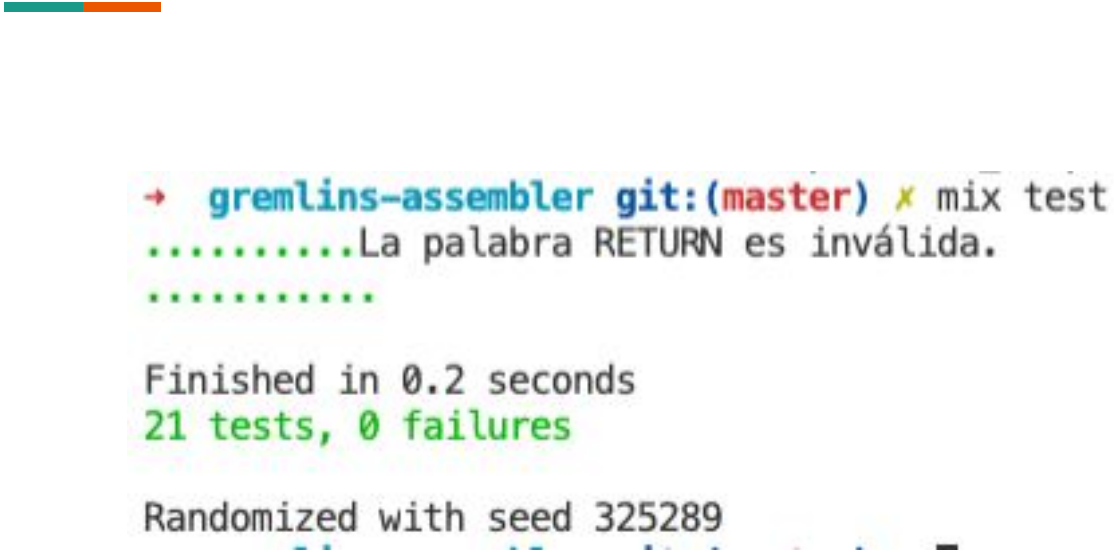
```
def codigo_gen(:bitewise_Reserveword, _, codigo,post_stack) do
  if List.first(post_stack) == "return" do
    codigo <> """
      not     %rax
    """
  else
    codigo <> """
      not     %rax
      push    %rax
    """
  end

end
```

Tests

```
test "Prueba 6 de Nora Sandler: Operador unario, complemento bit a bit de 0" do
  token_list = Lexer.scan_word(File.read!("test/bitwise_zero.c"), :no_output);
  assert  Parser.parse_token_list(elem(token_list, 1), :no_output) ==
    {:ok, {:program, "program",
           {:function, "main",
            {:return_Reserveword, "return",
             {:bitewise_Reserveword, "~", {:constant, 0, {}, {}}, {}}, {}}, {}}, {}}, {}}}
end
```

```
test "Prueba 7 de Nora Sandler: Operador unario, negación" do
   token_list = Lexer.scan_word(File.read!("test/negacion.c"), :no_output);
   assert  Parser.parse_token_list(elem(token_list, 1), :no_output) ==
     {:ok, {:program, "program",
            {:function, "main",
             {:return_Reserveword, "return",
              {:negation_Reserveword, "-", {:constant, 5, {}, {}}, {}}, {}}, {}}, {}}, {}}}
 end
```


gremlins-assembler git:(master) ✗ mix test
.....La palabra RETURN es inválida.
.....
Finished in 0.2 seconds
21 tests, 0 failures
Randomized with seed 325289

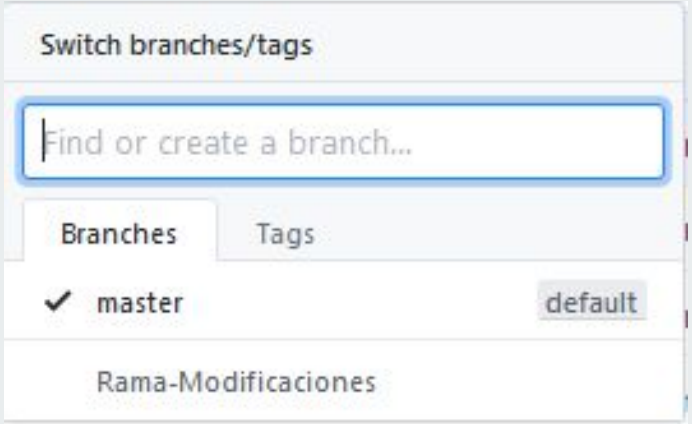
Changes in git

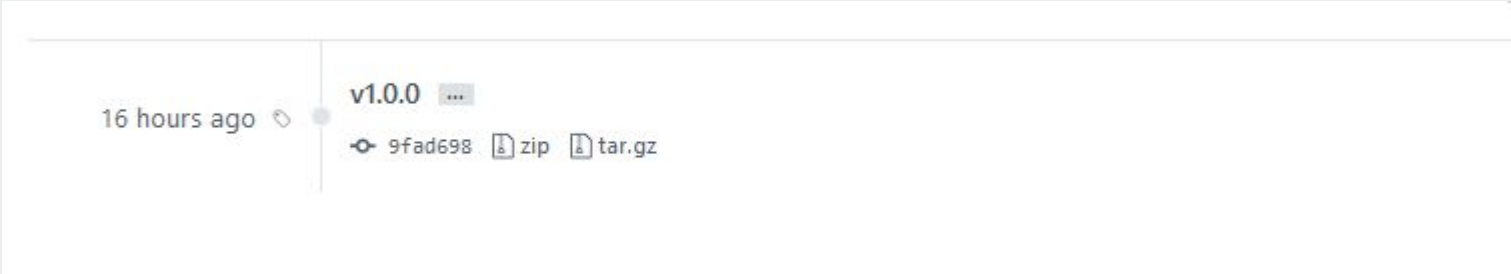
-Update in the master branch.

-Label creation in github called V1.0.0.

-Branch creation for changes and modifications.

Changes in git





Conclusions.

Deliverables and assigned dates for modifications need to be further improved.

Communication with the client must be more efficient and we must show more interest in knowing their points of view.

Improve the technical part for understanding elixir and the other tools used

Third Delivery

Barrientos Veana Luis Mauricio.

González Pacheco Leonardo Alonso.

Martínez Matías Joan Eduardo.

Rosales Romero Ricardo.

General changes

Four binary operators added.

- Addition
- Substraction
- Multiplication
- Division

Handle associativity and operator precedence

- Updates to functions in parser
- Updates to code generator

Adding binary operators to lexer

```
def lex_raw_tokens(program) when program != "" do
  {token, resto} =
  case program do
      "{" <> resto -> {:open_brace, resto}
      "}" <> resto -> {:close_brace, resto}
      "(" <> resto -> {:open_par, resto}
      ")" <> resto-> {:close_par, resto}
      ";" <> resto -> {:semicolon, resto}
      "return" <> resto -> {:return_Reserveword, resto}
      "int" <> resto -> {:int_Reserveword, resto}
      "main" <> resto -> {:main_Reserveword, resto}
      "-" <> resto -> {:negation_Reserveword, resto}
      "!" <> resto -> {:logicalNeg, resto}
      "~" <> resto -> {:bitewise_Reserveword, resto}
      "+" <> resto -> {:add_Reserveword, resto}
      "*" <> resto -> {:multiplication_Reserveword, resto}
      "/" <> resto -> {:division_Reserveword, resto}
```

Modifying parse_term to manage multiplication and division

```
def parse_term(tokens, last_op) do
  #envia el operador parseado con anterioridad por si ocurre un error
  [tokens, node_factor] = pars_factor(tokens, last_op); #oks
  case tokens do
    {:error, _} -> [tokens, ""]
    _ -> if List.first(tokens) == :multiplication_Reserveword or
            List.first(tokens) == :division_Reserveword do
            next_fact_term(tokens, node_factor)
          else #cuando no hay multiplicacion o division
            [tokens, node_factor];
          end
  end
end
```

Modifying next_fact_term to manage multiplication and division

```
def next_fact_term(tokens, node_factor)  do
  [tokens, operator] = parse_oper(tokens); #extrae el operador 1
  [tokens, next_factor] = pars_factor(tokens, operator) #extrae el operador 2

  # construccion del nodo con suma o resta
  [tokens, node_factor] = parse_bin_op(tokens, operator, node_factor, next_factor);
  #recursividad
  case tokens do
    {:error, _} -> [tokens, ""]
    _ -> if List.first(tokens) == :multiplication_Reserveword or
            List.first(tokens) == :division_Reserveword do
            next_fact_term(tokens, node_factor)
          else #cuando no hay multiplicacion o division
            [tokens, node_factor];
          end
  end
end
```

Changes in pars_factor

```
else
  case List.first(tokens) do
    {:constant, _} -> parse_constant(tokens, :constant)
    _ -> if (List.first(tokens)) == :add_Reserveword
           or (List.first(tokens)) == :multiplication_Reserveword
           or (List.first(tokens)) == :division_Reserveword do
           [{:error, "Error de sintaxis: Falta el primer operando antes de " <> dicc(List.first(tokens)) <> "."}, ""]
         else
           if last_op == :addition_Reserveword or last_op == :min_Reserveword
             or last_op == :multiplication_Reserveword
             or last_op == :division_Reserveword do
             [{:error, "Error de sintaxis: Falta el segundo operando después de " <> dicc(last_op) <> "."}, ""]
           else
             [{:error, "Error de sintaxis: Se esperaba una constante u operador y se encontró " <> dicc(List.first(tokens)) <> "."}, ""]
           end
         end
  end
end
```

Dictionary to convert reserved words to characters

```
def dicc(atom)do
    case atom do
        :int_Reserveword->"int"
        :main_Reserveword->"main"
        :open_par->"("
        :close_par->")"
        :open_brace->"{"
        :close_brace->"}"
        :logicalNeg->"!"
        :bitewise_Reserveword -> "~"
        :negation_Reserveword->"-"
        :min_Reserveword -> "-"
        :add_Reserveword -> "+"
        :return_Reserveword->"return"
        :semicolon->";"
        :multiplication_Reserveword->"*"
        :division_Reserveword->"/"
        _ -> "(empty)"
    end
end
```

Adding binary operators to code generator

```
def codigo_gen(:multiplication_Reserveword, _, codigo, _) do
    codigo <> """
      pop    %rcx
      imul   %ecx, %eax
      push   %rax
    """
end

def codigo_gen(:division_Reserveword, _, codigo, _) do
  codigo <> """
      push   %rax
      pop    %rcx
      pop    %rax
      xor    %edx, %edx
      idivl  %ecx
      push   %rax
  """
end
```

```
def codigo_gen(:min_Reserveword, _, codigo, _) do
  codigo <> """
      pop     %rcx
      sub     %rax, %rcx
      mov     %rcx, %rax
  """
end

def codigo_gen(:add_Reserveword, _, codigo, _) do
    codigo <> """
      pop    %rcx
      addl   %ecx, %eax
      push   %rax
    """
end
```

Handling binary expressions

```
def codigo_gen(:constant, value, codigo, post_stack) do
    if List.first(post_stack) == "+"
        or List.first(post_stack) == "-"
        or List.first(post_stack) == "*"
        or List.first(post_stack) == "/"
        or List.first(post_stack) == "~"
        or List.first(post_stack) == "!" do
        codigo <> """
            movl $#{value},%eax
        """
    else
        codigo <> """
            mov  $#{value}, %rax
            push    %rax
        """
    end
end
```

Some tests

```
test "Prueba 3-1 de Nora Sandler: Sin punto y coma" do
  token_list = Lexer.scan_word(File.read!("test/codigoc/sin_semicolo.c"), :no_output);
  assert  Parser.parse_token_list(elem(token_list, 1), :no_output) == {:error, "Error de sintáxis. Se esperaba ; y se encontró: }"}
end

test "Prueba 3-2 de Nora Sandler: Falta el primer operando." do
  token_list = Lexer.scan_word(File.read!("test/codigoc/falta_primer_oper.c"), :no_output);
  assert  Parser.parse_token_list(elem(token_list, 1), :no_output) == {:error, "Error de sintaxis: Falta el primer operando antes de +."}
end

test "Prueba 3-3 de Nora Sandler: Falta el segundo operando." do
  token_list = Lexer.scan_word(File.read!("test/codigoc/falta_seg_oper.c"), :no_output);
  assert  Parser.parse_token_list(elem(token_list, 1), :no_output) == {:error, "Error de sintaxis: Se esperaba una constante u operador y se encontró ;."}
end
```

Learning Obtained

We have realized that following a methodology when programming makes our interaction as a team more efficient, so for the last installment we will establish a stricter plan to improve all those details that are missing in the compiler.

The tests are important because with them we verify that the previous deliveries are correctly elaborated, and the modularization of the code is a broad advantage since errors can be easily detected.