# Compiler

## Jubilados
## Third Stage

**Project Manager**
Ruiz Aguilar Eduardo

**System Architect**
Rodriguez Garcia Dulce Coral

**System Tester**
Hernández Escobar Oswaldo

**System Integrator**
Aguilera Ortiz Alfredo

# Topics

- Requirements
  - Binary operators
- Code Modifications
  - Lexer
    Addition +
    Multiplication *
    Division /
  - Parser
  - Code Generator
- Test
  - Before
  - After

# Modifications

## Parser

```
def parse_expression([{next_token, num_line} | rest]) do
  term = parse_term([{next_token, num_line} | rest])
  {expression_node, term_rest} = term
  [{next_token, num_line} | rest] = term_rest
  case next_token do
    :add_operator ->
        subTree = %AST{node_name: :addition}
        parse_op = parse_expression(rest)
        {node,parse_rest} = parse_op
        [{next_token,num_line} | rest_op] = parse_rest
        {%{subTree | left_node: expression_node , right_node: node}, parse_rest}
    :neg_operator ->
        subTree = %AST{node_name: :substraction}
        parse_op = parse_expression(rest)
        {node,parse_rest} = parse_op
        [{next_token,num_line} | rest_op] = parse_rest
        {%{subTree | left_node: expression_node , right_node: node}, parse_rest}
    _ ->
      term
  end
end

def parse_term([{next_token, num_line} | rest]) do
  factor = parse_factor([{next_token, num_line} | rest])
  {expression_node, factor_rest} = factor
  [{next_token, num_line} | rest] = factor_rest
  case next_token do
    :mult_operator ->
        subTree = %AST{node_name: :multiplication}
        parse_op = parse_expression(rest)
        {node,parse_rest} = parse_op
        [{next_token,num_line} | rest_op] = parse_rest
        {%{subTree | left_node: expression_node , right_node: node}, parse_rest}
    :div_operator ->
        subTree = %AST{node_name: :division}
        parse_op = parse_expression(rest)
        {node,parse_rest} = parse_op
        [{next_token,num_line} | rest_op] = parse_rest
        {%{subTree | left_node: expression_node , right_node: node}, parse_rest}
    _ ->
      factor
  end
end
```

## Code Generator

```
def emit_code(:return, code_snippet, _) do
  """
      movl    #{code_snippet}, %eax
      ret
  """
end

def emit_code(:negation, code_snippet, _) do
  code_snippet <>
    """
        neg   %eax
    """
end

def emit_code(:bitwise, code_snippet, _) do
  code_snippet <>
    """
        not   %eax
    """
end

def emit_code(:logical_negation, code_snippet, _) do
  code_snippet <>
    """
        cmpl  #{code_snippet}, %eax
        movl  #{code_snippet}, %eax
        sete  %al
    """
end

def emit_code(:addition, _code_snippet, _) do
  """
        push  %rax
        pop   %rcx
        addl  %ecx, %eax
  """
end

def emit_code(:multiplication, _code_snippet, _) do
  """
        push  %rax
        pop   %rcx
        imul  %ecx, %eax
  """
end

def emit_code(:substraction, _code_snippet, _) do
  """
```

# Modifications

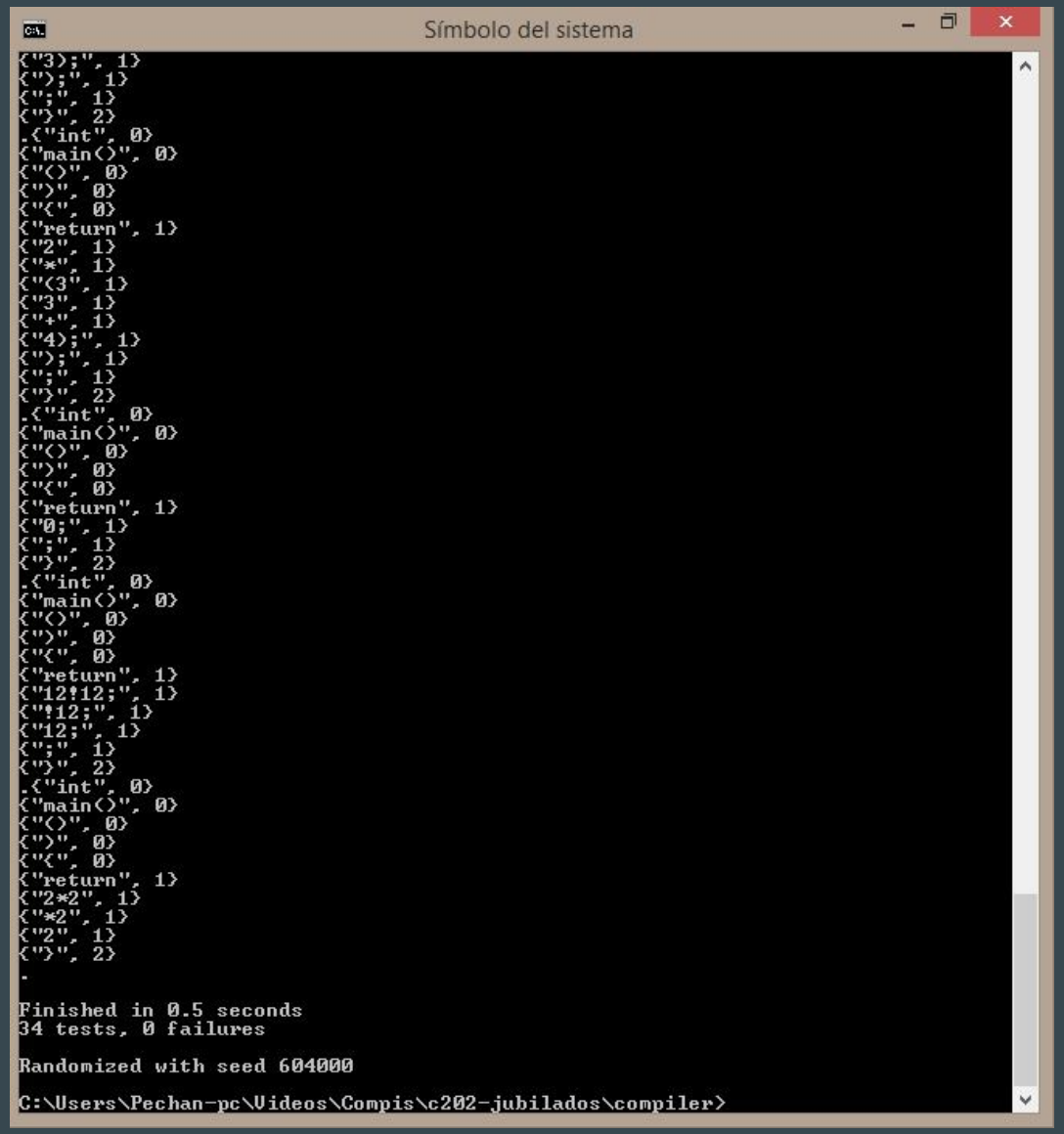
```elixir
def parse_factor([{next_token, num_line} | rest]) do
  case next_token do
    :open_paren ->
      if next_token == :open_paren do
        expression = parse_expression(rest)
        case expression do
            {{:error, error_message, num_line, next_token}, rest} ->
                {{:error, error_message, num_line, next_token}, rest}

            {exp_node, [{next_token,num_line} | rest]} ->
                if next_token == :close_paren do
                  {exp_node, rest}
                else
                  express = parse_expression(rest)
                  {node_expression,exp_rest} = expression
                  {node,[{next_token,num_line} | rest]} = exp_rest
                  {%AST{node_expression | left_node: node}, rest}
                end
        end
      else
        {{:error, "Error: factor '(' ",num_line,next_token}, rest}
      end
      :neg_operator ->
        parse_unary_op([{next_token, num_line} | rest])
      :bitwise_operator ->
        parse_unary_op([{next_token, num_line} | rest])
      :logical_neg_operator ->
        parse_unary_op([{next_token, num_line} | rest])
      {:constant, value} ->
        {%AST{node_name: :constant, value: value}, rest}
      _ ->
        {{:error, "Error: incomplete factor", num_line, next_token}, rest}
  end
end
```

```elixir
def parse_unary_op([{next_token, num_line}| rest]) do
  case next_token do
    :neg_operator ->
      if (hd rest) == {:neg_operator, num_line} do
        error_message =  "Error: can´t handle multiple operator in line"
        {{:error, error_message}, rest}
      else
        parse_unary = parse_factor(rest)
        {function_node,rest} = parse_unary
        case parse_unary do
          {{:error, error_message}, rest} ->
            {{:error, error_message}, rest}
          _ ->
            {%AST{node_name: :negation, left_node: function_node}, rest}
        end
      end
    :bitwise_operator ->
      parse_unary = parse_factor(rest)
      {function_node,rest} = parse_unary
      case parse_unary do
        {{:error, error_message}, rest} ->
            {{:error, error_message}, rest}
        _ ->
            {%AST{node_name: :bitwise, left_node: function_node}, rest}
      end
    :logical_neg_operator ->
      parse_unary = parse_factor(rest)
      {function_node,rest} = parse_unary
      case parse_unary do
        {{:error, error_message}, rest} ->
            {{:error, error_message}, rest}
        _ ->
            {%AST{node_name: :logical_negation, left_node: function_node}, rest}
      end
  end
end
```

# Tree Example

```
:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
{:ok,
 %AST{
   left_node: %AST{
     left_node: %AST{
       left_node: %AST{
         left_node: %AST{
           left_node: %AST{
             left_node: nil,
             node_name: :constant,
             right_node: nil,
             value: 1
           },
           node_name: :negation,
           right_node: nil,
           value: nil
         },
         node_name: :addition,
         right_node: %AST{
           left_node: %AST{
             left_node: nil,
             node_name: :constant,
             right_node: nil,
             value: 4
           },
           node_name: :multiplication,
           right_node: %AST{
             left_node: %AST{
               left_node: %AST{
                 left_node: nil,
                 node_name: :constant,
                 right_node: nil,
                 value: 4
               },
               node_name: :division,
               right_node: %AST{
                 left_node: nil,
```

# Test




SCANNER

# Test

Símbolo del sistema

```
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
:int_keyword
.:main_keyword
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
.:int_keyword
:main_keyword
:open_paren
:close_paren
:open_brace
.

Finished in 0.6 seconds
28 tests, 0 failures

Randomized with seed 420000

C:\Users\Pechan-pc\Videos\Compis\c202-jubilados\compiler>mix test test/Parser_te
st.exs_
```

# Parser