Project Scope
Fourth Delivery

Project Title: C Compiler on Elixir

Client: Ortigoza Márquez Norberto J.

Authors: Ruiz Aguilar Eduardo (Project Manager)

Aguilera Ortíz Alfredo (Integrator)

Hernández Escobar Oswaldo (Tester)

Rodríguez García Dulce Coral (Architect)

Due Date: MAY 29, 2020
Version:4 .0
Circulation: Internal

In this installment we´re going to add some boolean operators, also we're going to add more relational operators.

As said before, compared to the previous delivery, the same logic would be followed, so, by successfully implementing a grammar which supports simple binary operators like (*+, -, *, /*) the next delivery would be considered easier knowing that new operators work in the same way, the eight binary operators that were added will be shown below.

- Logical AND &&
- Logical OR ||
- Equal to ==
- Not equal to !=
- Less than <
- Less than or equal to <=
- Greater than >
- Greater than or equal to >=

As usual, to add support for new operators on the compiler, the changes must be done in the lexer, parser and code generator once again.

Lexing Modifications

As we know each operator it's a new token in the lexing, additionally will be added tokens like:

- AND &&
- OR ||
- Equal ==
- Not Equal !=
- Less than <
- Less than or equal <=
- Greater than >
- Greater than or equal >=

Parser modifications

For this stage we have more levels of precedence, so our grammar is bigger. Below are our binary operators but this time ordered from highest to lowest hierarchy.

- Multiplication & division (*, /)
- Addition & subtraction (+,-)
- Relational less than/greater than/less than or equal/greater than or equal (<, >,<=,>=)
- Relational equal/not equal (==, !=)
- Logical AND (&&)
- Logical OR (||)

At this point our grammar supports just the first two points from aboves hierarchy, the other grammar rules will also be added. Taking a look at the grammar proposed by Nora Sandler's blog, parsing must look like this.

```
<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <logical-and-exp> { "||" <logical-and-exp> }
<logical-and-exp> ::= <equality-exp> { "&&" <equality-exp> }
<equality-exp> ::= <relational-exp> { ("!=" | "==") <relational-exp> }
<relational-exp> ::= <additive-exp> { ("<" | ">" | "<=" | ">=") <additive-exp> }
<additive-exp> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/") <factor> }
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
<unary_op> ::= "!" | "~" | "-"
```

And in code, each one of the non terminals works as a function, the ones in bold represent the new ones that need to be added.

```
def parse_expression([{next_token, num_line} | rest]) do
   logical_and_expression = parse_logical_and_expression([{next_token, num_line} | rest])
   {expression_node, logical_and_expression_rest} = logical_and_expression
   [{next_token, num_line} | rest] = logical_and_expression_rest
   case next_token do
     :or_operator ->
         subTree = %AST{node_name: :or_op}
         parse_op = parse_logical_and_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}
     _ ->
       logical_and_expression
   end
 end

def parse_logical_and_expression([{next_token, num_line} | rest]) do
   equality_expression = parse_equality_expression([{next_token, num_line} | rest])
   {expression_node, equality_expression_rest} = equality_expression
   [{next_token, num_line} | rest] = equality_expression_rest
   case next_token do
     :and_operator ->
         subTree = %AST{node_name: :and_op}
         parse_op = parse_equality_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}
     _ ->
       equality_expression
   end
 end
```

```
def parse_equality_expression([{next_token, num_line} | rest]) do
   relational_expression = parse_relational_expression([{next_token, num_line} | rest])
   {expression_node, relational_expression_rest} = relational_expression
   [{next_token, num_line} | rest] = relational_expression_rest
   case next_token do
     :not_equal_operator ->
         subTree = %AST{node_name: :not_equal}
         parse_op = parse_relational_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}
     :equal_operator ->
         subTree = %AST{node_name: :equal}
         parse_op = parse_relational_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}
     _ ->
       relational_expression
   end
 end
```

```
def parse_relational_expression([{next_token, num_line} | rest]) do
  additive_expression = parse_additive_expression([{next_token, num_line} | rest])
  {expression_node, additive_expression_rest} = additive_expression
  [{next_token, num_line} | rest] = additive_expression_rest
  case next_token do
    :less_than_operator ->
        subTree = %AST{node_name: :less_than}
        parse_op = parse_additive_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}
    :greater_than_operator ->
        subTree = %AST{node_name: :greater_than}
        parse_op = parse_additive_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}
    :less_than_or_equal_operator ->
        subTree = %AST{node_name: :less_than_or_equal}
        parse_op = parse_additive_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}
    :greater_than_or_equal_operator ->
        subTree = %AST{node_name: :greater_than_or_equal}
        parse_op = parse_additive_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}
    _ ->
      additive_expression
  end
end
```

```
def parse_additive_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_term(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_term(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
```

Let's note that the name of <exp> is now called <additive-exp> since it refers to logical expressions that have lower priority.

Code Generator

As in the previous installment, the code generation will be the same

1. Calculate e1.

2. Push it onto the pile.

3. Calculate e2.

4. Pop the stack back into a register.

5. Perform the operation on e1 and e2.

Relational operators

As a priority we will handle the relational operators, unlike the NOT operator in second delivery, this time they return 1 for true results and 0 for false results. The C11 standard guarantees that evaluation && and || will cause a short circuit: if we know the result after evaluating the first clause, we do not evaluate the second clause.

Logical OR

To guarantee that logical OR short, we will have to jump over clause 2 when clause 1 is true. We will follow these steps to calculate e1 || e2:

1. Calculate e1

2. If the result is 0, skip to step 4.

3. Set EAX to 1 and jump to the end.

4. Calculate e2.

5. If the result is 0, set EAX to 0. Otherwise, set EAX to 1.

Logical AND

It will be almost identical to the logical OR, with the difference that we will short-circuit if e1 is 0.

Reference

https://norasandler.com/2017/12/28/Write-a-Compiler-4.html