

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 above's 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.

```

94 def parse_expression([next_token, num_line] | rest) do
95   logical_and_expression = parse_logical_and_expression([next_token, num_line] | rest)
96   {expression_node, logical_and_expression_rest} = logical_and_expression
97   [{next_token, num_line} | rest] = logical_and_expression_rest
98   case next_token do
99     :or_operator ->
100       subTree = %AST{node_name: :or_op}
101       parse_op = parse_logical_and_expression(rest)
102       {node, parse_rest} = parse_op
103       [{next_token, num_line} | rest_op] = parse_rest
104       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
105     ->
106       logical_and_expression
107   end
108 end
109
110 def parse_logical_and_expression([next_token, num_line] | rest) do
111   equality_expression = parse_equality_expression([next_token, num_line] | rest)
112   {expression_node, equality_expression_rest} = equality_expression
113   [{next_token, num_line} | rest] = equality_expression_rest
114   case next_token do
115     :and_operator ->
116       subTree = %AST{node_name: :and_op}
117       parse_op = parse_equality_expression(rest)
118       {node, parse_rest} = parse_op
119       [{next_token, num_line} | rest_op] = parse_rest
120       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
121     ->
122       equality_expression
123   end
124 end
125
126 def parse_equality_expression([next_token, num_line] | rest) do
127   relational_expression = parse_relational_expression([next_token, num_line] | rest)
128   {expression_node, relational_expression_rest} = relational_expression
129   [{next_token, num_line} | rest] = relational_expression_rest
130   case next_token do
131     :not_equal_operator ->
132       subTree = %AST{node_name: :not_equal}
133       parse_op = parse_relational_expression(rest)
134       {node, parse_rest} = parse_op
135       [{next_token, num_line} | rest_op] = parse_rest
136       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
137     :equal_operator ->
138       subTree = %AST{node_name: :equal}
139       parse_op = parse_relational_expression(rest)
140       {node, parse_rest} = parse_op
141       [{next_token, num_line} | rest_op] = parse_rest
142       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
143     ->
144       relational_expression
145   end
146 end

```

```

148 def parse_relational_expression([next_token, num_line] | rest) do
149   additive_expression = parse_additive_expression([next_token, num_line] | rest)
150   {expression_node, additive_expression_rest} = additive_expression
151   [{next_token, num_line} | rest] = additive_expression_rest
152   case next_token do
153     :less_than_operator ->
154       subTree = %AST{node_name: :less_than}
155       parse_op = parse_additive_expression(rest)
156       {node, parse_rest} = parse_op
157       [{next_token, num_line} | rest_op] = parse_rest
158       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
159     :greater_than_operator ->
160       subTree = %AST{node_name: :greater_than}
161       parse_op = parse_additive_expression(rest)
162       {node, parse_rest} = parse_op
163       [{next_token, num_line} | rest_op] = parse_rest
164       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
165     :less_than_or_equal_operator ->
166       subTree = %AST{node_name: :less_than_or_equal}
167       parse_op = parse_additive_expression(rest)
168       {node, parse_rest} = parse_op
169       [{next_token, num_line} | rest_op] = parse_rest
170       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
171     :greater_than_or_equal_operator ->
172       subTree = %AST{node_name: :greater_than_or_equal}
173       parse_op = parse_additive_expression(rest)
174       {node, parse_rest} = parse_op
175       [{next_token, num_line} | rest_op] = parse_rest
176       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
177   ->
178     additive_expression
179   end
180 end

182 def parse_additive_expression([next_token, num_line] | rest) do
183   term = parse_term([next_token, num_line] | rest)
184   {expression_node, term_rest} = term
185   [{next_token, num_line} | rest] = term_rest
186   case next_token do
187     :add_operator ->
188       subTree = %AST{node_name: :addition}
189       parse_op = parse_term(rest)
190       {node, parse_rest} = parse_op
191       [{next_token, num_line} | rest_op] = parse_rest
192       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
193     :neg_operator ->
194       subTree = %AST{node_name: :subtraction}
195       parse_op = parse_term(rest)
196       {node, parse_rest} = parse_op
197       [{next_token, num_line} | rest_op] = parse_rest
198       {%{subTree | left_node: expression_node, right_node: node}, parse_rest}
199   ->
200     term
201   end
202 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>