Faculty of Engineering,
Alexandria University,
Computer and Systems Engineering Department

Phase 2
Compiler
"Parser Generator"

Names:

Yomna Ahmed 71

Nada Ahmed 66

Marwan Tarek 61

Mohamed Elsabbagh 52

A description of the used data structures:

1. We depend mainly on HashMap data structure .
    - In the class **ParseGenerator,** we use two hashmaps:
      HashMap<String,NonTerminal> non_terminal_map
      HashMap<String,Terminal> terminal_map
      to hold the non terminals and the the terminals of the grammar with their key is the name of the terminal or non terminal and the value is a reference to it.

2. We also use ArrayList<> data structure.
    - In the **NonTerminal** class, each non terminal holds its production rules: ArrayList<ArrayList<Component>> where the component is either terminal or non terminal and each production rule is ArrayList<Component>.
    - In the **Pair** class to represent each production where the non terminal is mentioned in the grammar.
    - In the **ParseGenerator** class, we use two dummy arraylists
      ArrayList<Component> dummy_for_epsilon
      ArrayList<Component> dummy_for_synch
      To represent the synch and the epsilon production in the parse table.
      And ArrayList<Component>[][] for the parse table.

3. We also use Stack data structure.
    - In the **process_input()** function, Stack is used to put the productions in the stack from the reverse direction ,for example (S → Aab if we need to replace S by this production in the stack we put it as the following b a A).

Explanation of algorithms and techniques used:

- **Parse the grammar file:**

We read the grammar file line by line to get the production rules and save the terminals and non terminals of the grammar each into its corresponding map. With each non terminal having its productions saved as an arraylist of terminals and non terminals. And also it has the productions, in which it's referred to in toto, to help us compute the follow set of this non terminal.

- **Eliminating left recursion:**

We iterate over the non terminal map and eliminate the left recursion of each non terminal after we substitute with each preceding non terminal mentioned in the current non terminal's production rules.

To Eliminate the left recursion we use this method:

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

- **Left factoring:**
  We iterate over the non terminal map and eliminate the left factoring of each non terminal by check if a non terminal has **two** or more production rules with a common non-empty prefix

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

  And convert it to

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

- **Compute first set:**
  We iterate over the non terminal map and for each nonterminal check its productions :
  - If X a where 'a' is terminal.
    - 'a' is in FIRST(X).
  - If X epsilon
    - epsilon is in FIRST(X).
  - If X Y where 'Y' is nonterminal.
    - FIRST(Y)is in FIRST(X).
  - If X Y1 Y2 Y3 …… Yn.
    - If epsilon is in FIRST(Yj) where j = 1,2,…,i-1 then FRIST(Yi) is in FIRST(X).
    - If epsilon is in FIRST(Yj) where j = 1,2,…,n then epsilon is in FIRST(X).

- **Compute follow set:**

First, If S is the start symbol --> $ is in FOLLOW(S). Then, we keep track of productions that a non terminal was mentioned in for each non terminal in the array list of pairs where pair has a source non terminal (the non terminal on LHS) and array list of components that represents the production and apply rules:

1. if A --> aBC is a production rule --> everything in FIRST(C) is FOLLOW(B) except epsilon.
2. If (A --> aB is a production rule) or (A --> aBC is a production rule and epsilon is in FIRST(C))
   everything in FOLLOW(A) is in FOLLOW(B).

- **Construct parse table:**

Since we have a map containing all non terminals and another one for terminals, we create the parsing table as a 2D array of productions. We map each non terminal to a row and map each terminal to a column in this array. Then we start filling entries in parsing table by iterating over the nonterminal map in this way :

1. we get the corresponding row for nonterminal x .
2. for each production :
   1) If the first component is a terminal , we get the column corresponding to it and put the production in this cell.
   2) If the first component is a nonterminal , we get columns corresponding to the first set of this nonterminal and put the production in these cells.
   3) If nonterminal x has epsilon production , we get columns corresponding to terminals in its follow set and put dummy for epsilon in these cells.

4) if nonterminal x don't have epsilon production , we get columns corresponding to terminals in its follow set and put dummy for sync in these cells (if these cells were empty).

- **Parse the input program:**

This part is implemented by the process_input function at first stack is created and push in it **“$”** sign and the starting Non terminal symbol then loop until stack is empty and inside this loop we check the following :

- If the top of the stack is Terminal then we have to paths :
    1) topOfstack name the same as that of the next token then pop from the stack
    2) Else Report Error and in the next loop don’t take the next token
- Else when the top of the stack is non terminal then we get the production that fit with the token and the Terminal/ Non terminal in the stack and here there are 3 cases, if after looking in the parsing table we find that:
    1) there is “sync” then :
        - Report an Error
        - pop the stack and don’t take the next token
    2) if it is Empty then :
        - Report an Error
        - get next token
    3) Otherwise, pop the stack and take the productions of the component pushing from last to first .

Input files:

input_RegEx.txt - Notepad

File Edit Format View Help

```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: \=
{ if else while }
[; , \( \) { }]
addop: \+ | -
mulop: \* | /
```

input_grammar.txt - Notepad

File Edit Format View Help

```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT = DECLARATION
| IF
| WHILE
| ASSIGNMENT
# DECLARATION = PRIMITIVE_TYPE ‘ id’ ‘;’
# PRIMITIVE_TYPE = ‘int’ | ‘float’
# IF = ‘if’ ‘(’ EXPRESSION ‘)’ ‘{’ STATEMENT ‘}’ ‘else’ ‘{’ STATEMENT ‘}’
# WHILE = ‘while’ ‘(’ EXPRESSION ‘)’ ‘{’ STATEMENT ‘}’
# ASSIGNMENT = ‘id’ ‘=’ EXPRESSION ‘;’
# EXPRESSION = SIMPLE_EXPRESSION
 | SIMPLE_EXPRESSION ‘relop’ SIMPLE_EXPRESSION
# SIMPLE_EXPRESSION = TERM | SIGN TERM | SIMPLE_EXPRESSION ‘addop’ TERM
# TERM = FACTOR | TERM ‘mulop’ FACTOR
# FACTOR = ‘id’ | ‘num’ | ‘(’ EXPRESSION ‘)’
# SIGN = ‘+’ | ‘-’
```

Test program:

test.txt - Notepad

File Edit Format View Help

```
int x;
x = 5;
if (x > 2)
{
x = 0;
}
```

Output:

output_grammar.txt - Notepad

File Edit Format View Help

```
METHOD_BODY -> STATEMENT_LIST

STATEMENT_LIST -> PRIMITIVE_TYPE STATEMENT_LIST'_4

PRIMITIVE_TYPE -> int

MATCHING ' int ' ...

STATEMENT_LIST'_4 -> id STATEMENT_LIST'_4'_1

MATCHING ' id ' ...

STATEMENT_LIST'_4'_1 -> ; STATEMENT_LIST'_4'_1'_1

MATCHING ' ; ' ...

STATEMENT_LIST'_4'_1'_1 -> STATEMENT_LIST$

STATEMENT_LIST$ -> ASSIGNMENT STATEMENT_LIST$

ASSIGNMENT -> id assign EXPRESSION ;

MATCHING ' id ' ...

MATCHING ' assign ' ...

EXPRESSION -> num EXPRESSION'_7

MATCHING ' num ' ...

EXPRESSION'_7 -> epsilon

MATCHING ' ; ' ...

STATEMENT_LIST$ -> IF STATEMENT_LIST$

IF -> if IF'_1

MATCHING ' if ' ...

IF'_1 -> ( IF'_1'_1
```

output_grammar.txt - Notepad

File Edit Format View Help

```
IF'_1 -> ( IF'_1'_1

MATCHING ' ( ' ...

IF'_1'_1 -> EXPRESSION IF'_1'_1'_1

EXPRESSION -> id EXPRESSION'_5

MATCHING ' id ' ...

EXPRESSION'_5 -> relop EXPRESSION'_5'_1

MATCHING ' relop ' ...

EXPRESSION'_5'_1 -> num

MATCHING ' num ' ...

IF'_1'_1'_1 -> ) IF'_1'_1'_1'_1

MATCHING ' ) ' ...

IF'_1'_1'_1'_1 -> { IF'_1'_1'_1'_1'_1

MATCHING ' { ' ...

IF'_1'_1'_1'_1'_1 -> id IF'_1'_1'_1'_1'_1'_2

MATCHING ' id ' ...

IF'_1'_1'_1'_1'_1'_2 -> assign IF'_1'_1'_1'_1'_1'_2'_1

MATCHING ' assign ' ...

IF'_1'_1'_1'_1'_1'_2'_1 -> EXPRESSION IF'_1'_1'_1'_1'_1'_2'_1'_1

EXPRESSION -> num EXPRESSION'_7

MATCHING ' num ' ...

EXPRESSION'_7 -> epsilon
```

output_grammar.txt - Notepad

File Edit Format View Help

```
EXPRESSION'_7 -> epsilon

IF'_1'_1'_1'_1'_1'_2'_1'_1 -> ; IF'_1'_1'_1'_1'_1'_2'_1'_1'_1

MATCHING ' ; ' ...

IF'_1'_1'_1'_1'_1'_2'_1'_1'_1 -> } IF'_1'_1'_1'_1'_1'_2'_1'_1'_1'_1

MATCHING ' } ' ...
```

Ln 1, Col 1

The resultant stream of tokens for the example test program:

output_tokens - Notepad

File Edit Format View Help

```
int
id
;
id
assign
num
;
if
(
id
relop
num
)
{
id
assign
num
;
}
```