

Compiler

PHASE II

Names:

- Mark Philip.
- Peter Atef.
- Aliaa Othman
- Radwa Elmasry.

Used Data Structures:

Map:

map< string, vector < vector < string> > > productionsMap;

- Used to hold the production the key (string) production name and the value is vector of vector of string to hold that production go to which productions.

map< string, vector<string>> first;

- Used to hold the first of each nonterminal production the key is string (production name) and the value is vector of strings that hold the first of production.

map< string, vector<string>> follow;

- Used to hold the follow of each nonterminal production the key is string (production name) and the value is vector of strings that hold the follow of production.

map< pair<string,string>, vector<string> > parsing_table;

- Used to hold the parsing Table the key (string ,string) first string for row element and second one for column element and the values in vector of string to hold that production used to the transition.

Vector:

vector< string > nonTerminals;

- This vector to hold the nonterminals productions.

vector< string > terminals;

- This vector to hold the terminals productions.

Stack:

stack <string> parsing_stack;

- The stack used in parsing the input to produce the output.

All Algorithms and Techniques used:

1 – Read a production file CFG file and produce a map that hold a production name and what it followed by in every line of production and put it in a vector of vector of string.

```
void getProductions(vector<string> productions)
    for I from 0 to productionsSize do
        -split on space
        -get a string and check if it not | then
        insert it in a vector of string
        -repeat a previous step until find | character
        -create a new vector of string and add the
        string in it
```

2- Get the first nonterminal productions and add them to the first map.

```
void getFirst()
    for i from nonterminalsSize to 0
        -get the productions from productions map
        -send the non terminal name and productions to
        getProductionsForFirst(productions,
        nonTerminals[i])

getProductionsForFirst(productions, nonTerminals[i])
    for i from 0 to productionsSize
        -check if the first element of every vector is
        terminal add it to first vector
        -if it is nonterminal then get the first of it
        and add it to the first vector
        -then insert the production name and vector of
        string in the first map
```

```
void getFollow(int NTIndex);  
void populateFollow();
```

Used for the populating the Follow map:

1. the *populateFollow()* used to loop on the productions and the grammar strings and call *getFollow()* to get the follow in recursive way.
2. In *getFollow()* loop on all nonterminal and on the production on each one to search for the nonterminal that *populateFollow* function send it.

```
void createTable();
```

Used for creating the parsing table form the follow and the first victors:

1. For each nonterminal (row) in the grammar we loop on its first and follow to fill the table cells as we study in lecture.
2. After the loop finish w add the sync to the table where it should go.
3. After that we print the table to the output file.

3- Generation of output part:

We keep track of the stack and input and check the cases of the following pseudocode.

```
1 void Parser::parse_tokens()
2     int token_count = 0 // index of current input token
3     parsing_stack.push("$") //first push the dollar sign to the stack
4     parsing_stack.push(starting_symbol) //push starting symbol to the stack
5
6     string top_of_stack
7     string input
8     vector<string> table_entry
9
10    WHILE(parsing_stack.size() != 0)
11
12        input <= get_next_token(token_count) //get next token from lexical analyzer
13        top_of_stack <= parsing_stack.top() //get the top of stack
14
15        IF(top_of_stack == "$" && input == "$") //successful match //first case //both are $
16            >> print that input is accepted.
17            break from the while
18        ELSE IF(is_terminal(top_of_stack)) //case 2 //top of stack is terminal symbol
19            IF(input == top_of_stack) //input and top of stack are the same terminal
20                >> match the input with top of stack.
21                parsing_stack.pop() //pop it from the stack
22                token_count++ //to get the next input token
23            ELSE //they are terminals but of different symbols
24                >> print error missing character.
25                parsing_stack.pop() //pop it from the stack
26        ELSE IF(!is_terminal(top_of_stack)) //case 3 //top of stack is not terminal
27            get table entry from the table
28            parsing_stack.pop() //pop the non terminal from stack
29            IF(table_entry != "sync")
30                parsing_stack.push(table_entry) //push table entry reversed
31            ELSE
32                parsing_stack.pop() //pop from stack
33        ELSE
34            >> print illegal
35            token_count++
36
```

Screenshots of tests:

The screenshot displays a code editor with five open files. The first file, `CFG.txt`, contains a grammar definition for a simple language. The second file, `main.cpp`, is empty. The third file, `void getFirst()`, contains a list of error messages generated during parsing. The fourth file, `parsing_output.txt`, contains the output of the parsing process. The fifth file, `test.txt`, contains a test program.

```
1 # METHOD BODY = STATEMENT LIST
2 # STATEMENT LIST = STATEMENT STATEMENT LIST_dash
3 # STATEMENT LIST_dash = STATEMENT STATEMENT LIST_dash | 'lamda'
4 # STATEMENT = DECLARATION
5 | IF
6 | WHILE
7 | ASSIGNMENT
8 # DECLARATION = PRIMITIVE_TYPE 'id' ';'
9 # PRIMITIVE_TYPE = 'int' | 'float'
10 # IF = 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
11 # WHILE = 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
12 # ASSIGNMENT = 'id' '=' EXPRESSION ';'
13 # EXPRESSION_dash = SIMPLE_EXPRESSION EXPRESSION_dash
14 # EXPRESSION_dash = 'lamda' | 'relop' SIMPLE_EXPRESSION
15 # SIMPLE_EXPRESSION = TERM SIMPLE_EXPRESSION_dash | SIGN TERM SIMPLE_EXPRESSION_dash
16 # SIMPLE_EXPRESSION_dash = 'addop' TERM SIMPLE_EXPRESSION_dash | 'lamda'
17 # TERM = FACTOR TERM_dash
18 # TERM_dash = 'mulop' FACTOR TERM_dash | 'lamda'
19 # FACTOR = 'id' | 'num' | '(' EXPRESSION ')'
20 # SIGN = '+' | '-'
```

```
1 METHOD_BODY => STATEMENT LIST
2 STATEMENT LIST => STATEMENT STATEMENT_LIST_dash
3 STATEMENT => DECLARATION
4 DECLARATION => PRIMITIVE_TYPE id ;
5 PRIMITIVE_TYPE => float
6 match float
7 match id
8 error missing character ;
9 illegal STATEMENT LIST_dash
10 STATEMENT LIST_dash => STATEMENT STATEMENT_LIST_dash
11 STATEMENT => ASSIGNMENT
12 ASSIGNMENT => id assign EXPRESSION ;
13 match id
14 error missing character assign
15 error sync
16 illegal STATEMENT LIST_dash
17 STATEMENT LIST_dash => STATEMENT STATEMENT_LIST_dash
18 STATEMENT => ASSIGNMENT
19 ASSIGNMENT => id assign EXPRESSION ;
20 match id
21 error missing character assign
22 illegal EXPRESSION
23 illegal EXPRESSION
24 error sync
25 illegal STATEMENT LIST_dash
26 STATEMENT LIST_dash => STATEMENT STATEMENT_LIST_dash
27 STATEMENT => ASSIGNMENT
28 ASSIGNMENT => id assign EXPRESSION ;
29 match id
30 error missing character assign
31 illegal EXPRESSION
32 illegal EXPRESSION
33 error sync
34 illegal STATEMENT LIST_dash
35 STATEMENT LIST_dash => STATEMENT STATEMENT_LIST_dash
36 STATEMENT => WHILE
37 WHILE => while ( EXPRESSION ) { STATEMENT }
38 match while
39 match (
40 EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
41 SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
```

```
1 float sum , count ;
2 pass ++ ;
3 pass -- ;
4 while ( pass != 10 ) {
5   pass = pass + 1 ;
6 }
7 if ( mnt <= 0 ) {
8   count = count + 1.234 ;
9 }
10 else
11 {
12   sum = sum + mnt ;
13 }
```

void getFirst()pasring_output.txt

1 METHOD_BODY => STATEMENT_LIST
2 STATEMENT_LIST => STATEMENT STATEMENT_LIST_dash
3 STATEMENT => DECLARATION
4 DECLARATION => PRIMITIVE_TYPE id ;
5 PRIMITIVE_TYPE => int
6 match int
7 match id
8 error missing character ;
9 illegal STATEMENT_LIST_dash
10 STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
11 STATEMENT => ASSIGNMENT
12 ASSIGNMENT => id assign EXPRESSION ;
13 match id
14 error missing character assign
15 illegal EXPRESSION
16 EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
17 SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
18 TERM => FACTOR TERM_dash
19 FACTOR => id
20 match id
21 illegal TERM_dash
22 illegal TERM_dash
23 TERM_dash => lamda
24 SIMPLE_EXPRESSION_dash => lamda
25 EXPRESSION_dash => lamda
26 match ;
27 STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
28 STATEMENT => WHILE
29 WHILE => while (EXPRESSION) { STATEMENT }
30 match while
31 match (
32 EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
33 SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
34 TERM => FACTOR TERM_dash
35 FACTOR => id
36 match id
37 TERM_dash => lamda
38 SIMPLE_EXPRESSION_dash => lamda
39 EXPRESSION_dash => relop SIMPLE_EXPRESSION
40 match relop
41 SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash

test.txttest_2.txt

1 int sum1 ,count,pass,mnt;
2 while(pass!=10)
3 {
4 pass=pass+1&
5 }
6 if(count==0)
7 mnt=10;
8 else
9 mnt=30;
10 |

void getFirst()pasring_output.txt

1 METHOD_BODY => STATEMENT_LIST
2 STATEMENT_LIST => STATEMENT STATEMENT_LIST_dash
3 STATEMENT => DECLARATION
4 DECLARATION => PRIMITIVE_TYPE id ;
5 PRIMITIVE_TYPE => int
6 match int
7 match id
8 match ;
9 STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
10 STATEMENT => ASSIGNMENT
11 ASSIGNMENT => id assign EXPRESSION ;
12 match id
13 match assign
14 EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
15 SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
16 TERM => FACTOR TERM_dash
17 FACTOR => num
18 match num
19 TERM_dash => lamda
20 SIMPLE_EXPRESSION_dash => lamda
21 EXPRESSION_dash => lamda
22 match ;
23 STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
24 STATEMENT => IF
25 IF => if (EXPRESSION) { STATEMENT } else { STATEMENT }
26 match if
27 match (
28 EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
29 SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
30 TERM => FACTOR TERM_dash
31 FACTOR => id
32 match id
33 TERM_dash => lamda
34 SIMPLE_EXPRESSION_dash => lamda
35 EXPRESSION_dash => relop SIMPLE_EXPRESSION
36 match relop
37 SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
38 TERM => FACTOR TERM_dash
39 FACTOR => num
40 match num
41 TERM_dash => lamda

test.txt

1 int x;
2 x = 5;
3 if (x > 2)
4 {
5 x = 0;
6 }
7 |

ed (1.6

Bonus Part:

A description of the used data structures :

Class Bonus :

Vectors :

- vector <pair< string, vector < vector < string>>>> productions_vector**
a vector holds the productions which is processed to eliminate left recursion and factoring .
- vector <pair< string, vector < vector < string>>>> temp_productions**
used to hold the new productions which are added during the elimination of left recursion and factoring.
- vector<string> non_terminals**
this vector holds non terminals symbols

Explanation of all algorithms and techniques used :

- 1- the file is parsed using the same technique and the productions is pushed into productions_vector.
- 2- iterating over this vector , for each production do the following to eliminate the left recursion :
 - if any production has the property that the non terminal in the left side of the production is existed in the left side of any of the right side terms which is separated by “|” if exists then left recursion is detected .
 - the rule says that if **A -> A alpha | beta** then to eliminate left recursion do the following :
A → beta A_dash
A_dash → alpha A_dash | lamda
 - this is done by : 1- take the other terms which don't have left recursion concatenate A_dash in the
end of each term then edit the old production to have this new right hand side.
2- make new production which its left side is A_dash and its right side is the terms followed the term “A” in the original production but before finding any “|”, then push it in temp_production vector.

3- iterating over the new production which are generated from eliminating left recursion, for each production do the following to eliminate left factoring :

- iterate over the right hand sides terms to find if there are any repeated terms which can be taken as common factor.

- if there exist the rule says that : $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$ then do the following :

$A \rightarrow \alpha A_dash$

$A_dash \rightarrow \beta_1 \mid \beta_2$

- this is done by : 1- edit the original production right hand side to be the common factor concatenated with A_dash in the production_vector.

terms
factor 2- make new production with left side A-dash and right side the remaining
after deleting the common factor from them, if term had only the common
then it became lamda after taking the common factor.