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();

<u>Used for the populating the Follow map:</u>

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();

<u>Used for creating the parsing table form the follow and the first victors:</u>

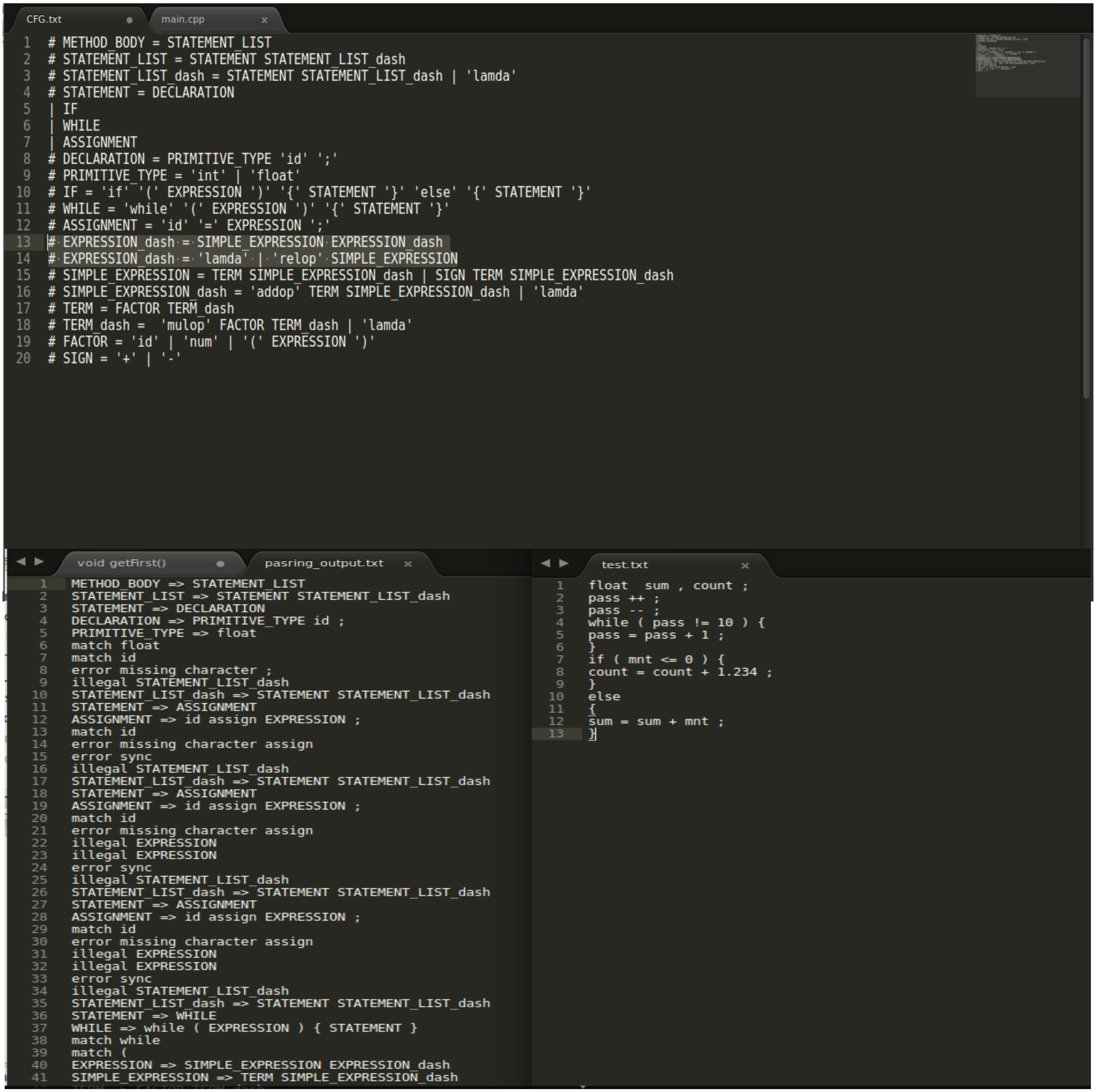
1. For each nonterminal (row) in the grammar we loop on its first and follow to fill the table cells as we study in lecture.
2. After the loop finish w add the sync to the table where it should go.
3. After that we print the table to the output file.

3- Generation of output part:

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

```
void Parser::parse_tokens()
    int token_count = 0 // index of current input token
    parsing_stack.push("$") //first push the dollar sign to the stack
    parsing_stack.push(starting_symbol) //push starting symbol to the stack

    string top_of_stack
    string input
    vector<string> table_entry

    WHILE(parsing_stack.size() != 0)

        input <= get_next_token(token_count) //get next token from lexical analyzer
        top_of_stack <= parsing_stack.top() //get the top of stack

        IF(top_of_stack == "$" && input == "$")  //successful match //first case //both are $
            >> print that input is accepted.
            break from thw while
        ELSE IF(is_terminal(top_of_stack)) //case 2 //top of stack is terminal symbol
            IF(input = top_of_stack) //input and top of stack are the same terminal
                >> match the input with top pf stack.
                parsing_stack.pop() //pop it from the stack
                token_count ++ //to get the next input token
            ELSE //they are terminals but of different symbols
                >> print error missing character.
                parsing_stack.pop() //pop it from the stack
        ELSE IF(!is_terminal(top_of_stack)) //case 3 //top of stack is not terminal
                get table entry from the table
                parsing_stack.pop() //pop the non terminal from stack
                IF(table_entry != "sync")
                    parsing_stack.push(table_entry) //push table entry reversed
                ELSE
                    parsing_stack.pop() //pop from stack
        ELSE
             >> print illegal
              token_count++
```

Screenshots of tests:



pasring_output.txt

```
METHOD_BODY => STATEMENT_LIST
STATEMENT_LIST => STATEMENT STATEMENT_LIST_dash
STATEMENT => DECLARATION
DECLARATION => PRIMITIVE_TYPE id ;
PRIMITIVE_TYPE => int
match int
match id
error missing character ;
illegal STATEMENT_LIST_dash
STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
STATEMENT => ASSIGNMENT
ASSIGNMENT => id assign EXPRESSION ;
match id
error missing character assign
illegal EXPRESSION
EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
TERM => FACTOR TERM_dash
FACTOR => id
match id
illegal TERM_dash
illegal TERM_dash
TERM_dash => lamda
SIMPLE_EXPRESSION_dash => lamda
EXPRESSION_dash => lamda
match ;
STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
STATEMENT => WHILE
WHILE => while ( EXPRESSION ) { STATEMENT }
match while
match (
EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
TERM => FACTOR TERM_dash
FACTOR => id
match id
TERM_dash => lamda
SIMPLE_EXPRESSION_dash => lamda
EXPRESSION_dash => relop SIMPLE_EXPRESSION
match relop
SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
```

test.txt

```
int sum1    ,count,pass,mnt;
    while(pass!=10)
{
pass=pass+1&
}
if(count==0)
mnt=10;
else
mnt=30;
```

pasring_output.txt

```
METHOD_BODY => STATEMENT_LIST
STATEMENT_LIST => STATEMENT STATEMENT_LIST_dash
STATEMENT => DECLARATION
DECLARATION => PRIMITIVE_TYPE id ;
PRIMITIVE_TYPE => int
match int
match id
match ;
STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
STATEMENT => ASSIGNMENT
ASSIGNMENT => id assign EXPRESSION ;
match id
match assign
EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
TERM => FACTOR TERM_dash
FACTOR => num
match num
TERM_dash => lamda
SIMPLE_EXPRESSION_dash => lamda
EXPRESSION_dash => lamda
match ;
STATEMENT_LIST_dash => STATEMENT STATEMENT_LIST_dash
STATEMENT => IF
IF => if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
match if
match (
EXPRESSION => SIMPLE_EXPRESSION EXPRESSION_dash
SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
TERM => FACTOR TERM_dash
FACTOR => id
match id
TERM_dash => lamda
SIMPLE_EXPRESSION_dash => lamda
EXPRESSION_dash => relop SIMPLE_EXPRESSION
match relop
SIMPLE_EXPRESSION => TERM SIMPLE_EXPRESSION_dash
TERM => FACTOR TERM_dash
FACTOR => num
match num
TERM_dash => lamda
```

test.txt

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

<u>Bonus Part:</u>

<u>A description of the used data structures :</u>

<u>Class Bonus :</u>

<u>Vectors :</u>

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

<u>Explanation of all algorithms and techniques used :</u>

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 → alpha beta1 | alpha beta2** then do the following :

A → alpha A_dash
A_dash → beta1 | beta 2

- this is done by : 1- edit the original production right hand side to be the common factor concateneted with A_dash in the production_vector.
2- make new production with left side A-dash and right side the remaining terms after deleting the common factor from them, if term had only the common factor then it became lamda after taking the common factor.