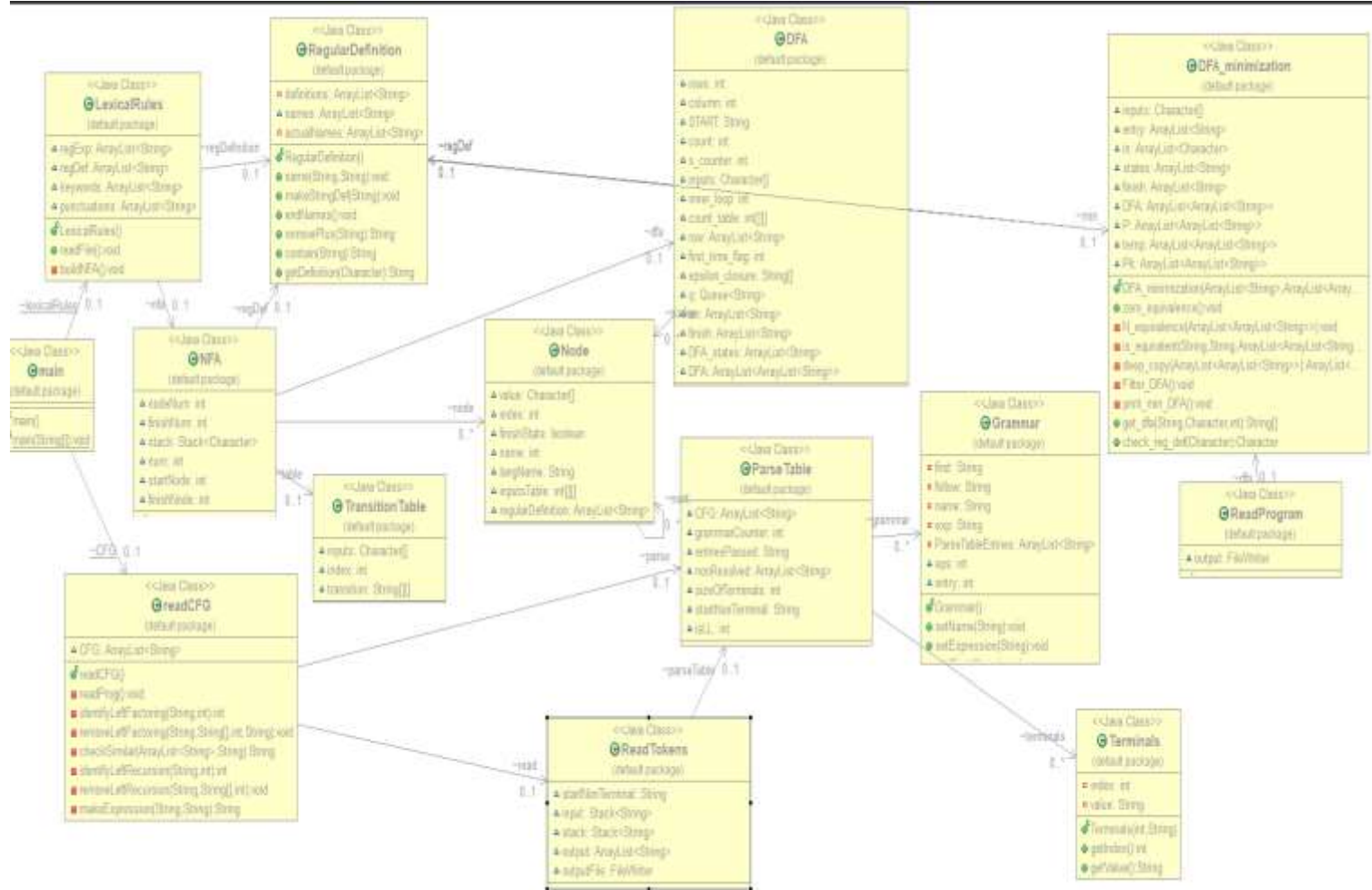


PLT

Phase Two Report

Mayar El Mahdy	4639
Al Zahraa Emara	4558

- **Class Diagram Figure:**



The class diagram consists of both classes used in phase one and two.

- **Data Structures:**

**Stack:** It was used in phase two when applying the parse table and the tokens (from phase one as input), to determine if it is accepted to the CFG or not. (In **ReadTokens** class)

**ArrayList :** It was used several times

-In class **ReadCFG**, an arraylist is used to store each CFG read from the input file.

- In class **ReadTokens**, an arraylist was used to store the output of terminals that were accepted by the CFG.

- **Algorithms and techniques:**

-**First:**

1-Read the CFG from bottom to up.

2-Split it each time we see ( | )

3- for loop on each definition

-If this definition begins with a terminal ( ' ) then it's first is this terminal.

-Else:

- If this definition is ( $\sim$ ) i.e.: epsilon then put epsilon in the first

-If this definition is nonterminal, then get the first of this nonterminal

There are two possibilities:

1- If this non terminal's first doesn't contain epsilon, then continue.

2-If this non terminal's first contains epsilon, then add epsilon to it's first.

```

for(int i=0;i<firstSplits.length;i++)
{
    //if it is a nonterminal add it to the first
    String temp = firstSplits[i];

    temp=temp.replaceAll("( +)"," ");
    String getF[] = temp.split(" ");
    if(!temp.equals(" ")) {
        if(getF.length<=1)
            temp = getF[0];
        else
            temp = getF[1];
        if(temp.compareTo("~")==0)
            grammar[grammarCounter].addFirst("~");
        else if(grammar[grammarCounter].getName() != temp)
        {
            if(Character.toString(temp.charAt(0)).compareTo("'')== 0)// it is a terminal
            {
                splitGrammar = temp.split("\\'");
                grammar[grammarCounter].addFirst(splitGrammar[1]);
            }
            else//non terminal
            {
                splitGrammar = temp.split("\\'"); //make sure there is no terminal without space
                temp = splitGrammar[0];
                if(replaceChanges(grammarCounter,temp)--1)
                {
                    firstSplits[i] = firstSplits[i].replace(temp, "");
                    firstSplits[i]=firstSplits[i].replaceAll("( +)"," ");
                    if(firstSplits[i].length()>1) {
                        i--;
                    }
                    else if(firstSplits[i].compareTo(" ") == 0)
                        grammar[grammarCounter].addFirst("~");
                }
                else if (replaceChanges(grammarCounter,temp)--1) {
                    nonResolved.add(Integer.toString(grammarCounter));
                }
            }
        }
    }
}

```

Figure: CalculateFirst, for loop.

## -Follow :

- 1-Read the CFG from top to bottom
- 2- Split it each ( | ) , and check the occurrence of the CFG name in all the CFGs.
- 3- When there is a match , Split on the occurrence of it's name then loop :
  - If there comes a terminal after it's name then add it to follow.
  - If there comes a non terminal after it , then add the non terminal's first
    - If the first contains epsilon then you need to remove epsilon and replace the non terminal's place and calculate the follow once more .
    - If it doesn't contain epsilon then simply add the non terminal's follow
  - If there is nothing after it's name then it's follow is the follow of the CFG it's at

```

for(int j=1;j<split.length;j++) {
    found = split[j];
    if(Character.toString(found.charAt(0)).compareTo("'")== 0)// it is a terminal
    {
        String splitGrammar[] = found.split("\\'");
        grammar[index].addFollow(splitGrammar[1]);
    }
    else if(found!=" " && Character.toString(found.charAt(0)).compareTo("|")!=0) {
        found=found.replaceAll("( +)"," ");
        String splitIt[] = found.split(" ");
        //splitIt[0] = " " + splitIt[0];
        String first = grammar[findIndex(splitIt[0])].getFirst();
        if(first.contains("ε")) { //if there is an epsilon then remove this non ter
            first = first.replace("ε", "");
            //then we remove the found non terminal
            String replace =found;
            replace = replace.replace(check, "");
            replace=replace.replaceAll("( +)"," ");
            grammar[index].addFollow(first);

            calculateFollow(index,found,splitIt[0] ,i);
        }
        grammar[index].addFollow(first);
    }
    else // take the follow of the grammar you are at
    {
        if(grammar[i].getFollow().equals(" ")) {
            nonResolved.add(Integer.toString(index));
        }
        else
            grammar[index].addFollow(grammar[i].getFollow());
    }
}
}

```

Figure: CalculateFollow, for loop.

- Functions Explanation:

-Left Factoring:

1-Identify it:

```
private int identifyLeftFactoring(String exp,int j)
{
    String split[] = exp.split("::=");
    String name = split[0];
    String expression = split[1];
    ArrayList<String> store = new ArrayList<String>();
    String[] split2 = expression.split("\\|");
    for(int i=0;i<split2.length;i++)
    {
        String similar = checkSimilar(store,split2[i]);
        if(!similar.equals(" "))
        {
            System.out.println("There is left factoring in " + exp);
            name = name.replace("#", "");
            removeLeftFactoring(name,split2,j , similar);
            return 1;
        }
        store.add(split2[i]);
    }
    return 0;
}
```

This function is used to identify if the grammar has Left factoring or not, so it begins by splitting the CFG with the name and the definition, then checks if there are any similar terminals/nonterminals between the OR

Ex: # A ::= 'a' B | 'a' C ,, there is left factoring , similar = 'a'

The array list called **store** stores the definitions that were checked before so we can check for I in the function **checkSimilar**

2- check for similarities:

```
    }  
    private String checkSimilar(ArrayList<String> store,String exp)  
    {  
        String[] compare = exp.split(" ");  
  
        compare[1] = " "+compare[1] + " ";  
        for(int i=0;i<store.size();i++)  
        {  
            String temp = store.get(i);  
            if(temp.startsWith(compare[1]))  
            {  
                temp = temp.replace(compare[1], "");  
  
                if(temp.startsWith(compare[2]))  
                    return compare[1] + compare[2] + " ";  
                return compare[1];  
            }  
        }  
        return " ";  
    }  
}
```

This function checks for similarities between the previous and the definition that we have now, so we check for max of the first two terms if they are similar or not.

First, we check **compare [1]** if there is a match! then check **compare [2]**

Return the common term.

Else return blank String – no similarity —

If there are similarities, then go to function **removeLeftFactoring**

### 3-Remove Left Factoring:

```
private void removeLeftFactoring(String name , String[] exp,int j ,String similar)
{
    String name2 = " " + name.replace(" ", "") + "DASH" + " ";
    String newExp = ""; //Store for new gxp
    String temp = ""; //Store for the exp
    int eps=0;
    //removes left factoring for common |
    for(int i=0;i<exp.length;i++)
    {
        if(exp[i].startsWith(similar))
        {
            //new gxp
            exp[i] = exp[i].replaceFirst(similar, " ");
            if(exp[i].equals(" "))
                eps=1;
            else
                newExp = newExp + "|" + exp[i];
        }
        else {
            //add to the old gxp
            temp = temp + "|" + exp[i];
        }
    }
    temp = temp + "|" + " "+similar + name2 ;
    temp = temp.replaceAll("\\s+", " ");
}
```

```
if(eps==1)
    newExp = newExp + "|" + " ~ ";
newExp = newExp.replaceFirst("\\|", "");
temp = temp.replaceFirst("\\|", "");
newExp = newExp.replaceAll("\\s+", " ");
String new1 = makeExpression(name,temp);
String new2 = makeExpression(name2 , newExp);
if(j==1) {
    CFG.add(new1);
    CFG.add(new2);
}
else {
    CFG.set(j, new1);
    j++;
    if(CFG.size() == j)
        CFG.add(new2);
    else {
        CFG.set(j, new2);
    }
}
}
```

This function is responsible for removing the left factoring and making two new CFG expressions rather than one.



The algorithm is rather simple, it splits the CFG when it sees (|) then checks if the similar String matches it, if yes then add in **newExpression** the expression without the common similar

Else add in the expression normally to **temp**. At the end you will have two CFG expressions.

Temp will take the name of the original CFG

New expression will take the name of the original CFG + add "DASH" to it .

**-Left Recursion:**

**1-Identify it:**

```
private int identifyLeftRecursion(String exp,int j)
{
    String[] split = exp.split("::=");

    String compare = split[0];
    String expression = split[1];

    compare = compare.replace("#", "");
    //split the expression when (|)
    String[] split2 = expression.split("\\|");
    for(int i=0;i<split2.length;i++)
    {
        String temp = split2[i];

        if(temp.startsWith(compare))
        {
            System.out.println("There is a left recursion in " + exp);
            removeLeftRecursion(compare,split2,j);
            return 1;
        }
    }
    return 0;
}
```

Identify the left recursion by checking if the name of CFG occurs as the start of the definition, split the definition each (|) and check the start String .

## 2-Remove Left Recursion:

Removing the left recursion algorithm is to take the splitting string (|) then checking each String if it starts with the name of the CFG then add it to the new Expression (after removing the occurrence of its name)

Else then add it to temp and also add the new name to it (new name is original CFG name + DASH)

Then add the new Expression and temp to the CFG as we did in the left factoring.

```
private void removeLeftRecursion(String name,String[] exp,int j)
{
    String temp = "";
    String newExp = "";
    String name2 = name.replace(" ", "");
    name2 = " " + name2 + "DASH" + " ";
    for(int i=0;i<exp.length;i++)
    {
        if(exp[i].startsWith(name))
        {
            exp[i] = exp[i].replace(name, " ");
            newExp = newExp+"|" + exp[i] + name2 + " ";
        }
        else {
            temp =temp+ "|" + exp[i]+ name2+ " ";
        }
    }
}
```

```
newExp = newExp.replaceFirst("\\|", "");
temp = temp.replaceFirst("\\|", "");

String new1 = makeExpression(name,temp);
newExp = newExp + "|" ~ " ";
String new2 = makeExpression(name2 , newExp);
if(j== -1) {
    CFG.add(new1);
    CFG.add(new2);
}
else {
    CFG.set(j, new1);
    j++;
    if(CFG.size() == j)
        CFG.add(new2);
    else {
        CFG.set(j, new2);
    }
}

}
```

## -makeParseTable:

This function uses the first and follow that was calculated for each CFG to build the parse table, Each grammar has a class called **Grammar**, each have an entry to the first, follow and the parse table entries.

There will be two for loops, the outer loop is for CFG the inner will be the terminals, so for building the parse table we will build it line by line.

In inner loop :

- We'll get the first of this CFG then check if there is already an entry in this position –If there is an entry so it is NOT a LL(1) grammar –else It will add an entry to the table.
- If the first has epsilon(~) then we see the follow, and add entries **NameOfCFG -> ~** where (~) is epsilon –Check if there was already an entry in the table if yes then it is NOT a LL(1) grammar.

```
private void makeParseTable()
{
    checkTerminals();
    initializeGrammarEntries();
    System.out.println("*****THE TABLE*****");
    //if multiple values then print error, return --Not LL(1)--
    for(int i=grammar.length-1;i>-1;i--)
    {
        for(int j=0;j<sizeOfTerminals;j++)
        {
            String term = terminals[j].getValue();
            if(grammar[i].getFirst().contains(term)) {

                String expression = grammar[i].getExpression();
                int checkError = addEntry(expression,term,i,j);
                if(checkError == 1)
                {
                    System.out.println("This is NOT a LL(1) Grammar");
                    isLL=0;
                    return;
                }
            }
            if(grammar[i].getFirst().contains("~"))
            {
                //take the follow
                if(grammar[i].getFollow().contains(term))
                {
                    String expression = grammar[i].getName() + "->" + "~";
                    int checkError = grammar[i].addEntry(terminals[j].getIndex(), expression);
                    if(checkError == 1)
                    {
                        System.out.println("This is NOT a LL(1) Grammar");
                        isLL=0;
                        return;
                    }
                }
            }
        }
    }
}
```

## -Grammar Class :

As stated above each CFG has an entry in Grammar class , so there are three functions responsible for the **parse table**

```
    }  
    public void initializeParseTableEntries(int sizeOfTerminals)  
    {  
        for(int i=0;i<sizeOfTerminals;i++)  
            ParseTableEntries.add("none");  
    }  
    public int addEntry(int index,String exp)  
    {  
        if(ParseTableEntries.get(index).equals("none")) {  
            ParseTableEntries.set(index, exp);  
            return 0; // no error  
        }  
        return 1; //if there is an error  
    }  
    public String getEntry(int index)  
    {  
        return ParseTableEntries.get(index);  
    }  
}
```

**1-initializeParseTableEntries** : This is used to fill up the ArrayList that will hold the entries to the parse table so it initialize them with 'none' to indicate that they are empty.

**2-addEntry** : This is used to fill up the entry to the parse table to this CFG so it first checks if it is empty then set it with the expression given else there is an error (Not a LL(1) grammar)

**3-getEntry** : This function is used to get the entries to the parse table so simply give the index to a specific terminal and it returns it's expression to this terminal.

## - Stack tracking:

We have two stacks the input line stack and the normal stack, this function takes the peek of both stacks as arguments, then checks if the stack peek is a nonterminal if yes it compares it with the input stack and pops both if equal, else panic mode error is printed.

If the peek of the stack is an expression, we call a function to bring its output from the parse table according to our input (peek of the input stack).

If the output brought from the table equals "none" i.e. table cell is empty for this output, panic mode error is activated, and a suitable error message is printed.

If the output equals epsilon we just pop the stack and return.

After that, if we reach this point it means that the output brought from the parse table is valid, so pop the stack and insert the replacement expressions brought from the table.

Lastly, we print the new stack and write it to the output file.

```
private void track_stack(String st, String ip) throws IOException
{
    if(st.contains(" "))
    {
        ip = ip.replace(" ", "");
        st = st.replace(" ", "");
        if(st.replaceAll(" ", "").equals(ip))
        {
            stack.pop();
            input.pop();
            System.out.print(stack + "\t\t\t");
            System.out.println(input);

            writeOutputFile(stack, "", input);
            output.add(ip);
            return;
        }
        else
        {
            String Err = "Error: Missing " + st + " inserted";
            //output.add(Err);
            output.add(st.replaceAll(" ", ""));
            System.err.println(Err);
            stack.pop();

            System.out.print(stack + "\t\t\t");
            System.out.println(input);
            writeOutputFile(stack, Err, input);
            return;
        }
    }

    String out = parseTable.getExpression(ip.replaceAll(" ", ""), st);
```

```
if(out.replaceAll(" ", "").equals("none"))
{
    if(!ip.equals("$"))
    {
        String Err = "Error: Illegal " + st + " discard " + ip;
        System.err.println("Error: Illegal " + st + " discard " + ip);
        writeOutputFile(stack, Err, input);
        input.pop();
    }
    else {
        String Err = "Error: Illegal " + st;
        System.err.println("Error: Illegal " + st);
        writeOutputFile(stack, Err, input);
        stack.pop();
    }
    System.out.print(stack + "\t\t\t");
    System.out.println(input);
    writeOutputFile(stack, "", input);
    return;
}
if(out.contains("~")){
    stack.pop();
    System.out.print(stack + "\t\t\t");
    System.out.println(input);
    writeOutputFile(stack, "", input);
    return;
}
String[] temp = out.split(" ");
stack.pop();
for(int i=temp.length-1; i>=0; i--)
{ stack.push(temp[i]);}

System.out.print(stack + "\t\t\t");
System.out.println(input);
writeOutputFile(stack, "", input);
```

## • Assumptions:

1-The CFG in CFG.txt are all separated by spaces, each line should end with a space.

Ex: # METHOD\_BODY ::= STATEMENT\_LIST

2- If there is a left recursion then it would be **direct** , Left factoring is applied to at most two commons .

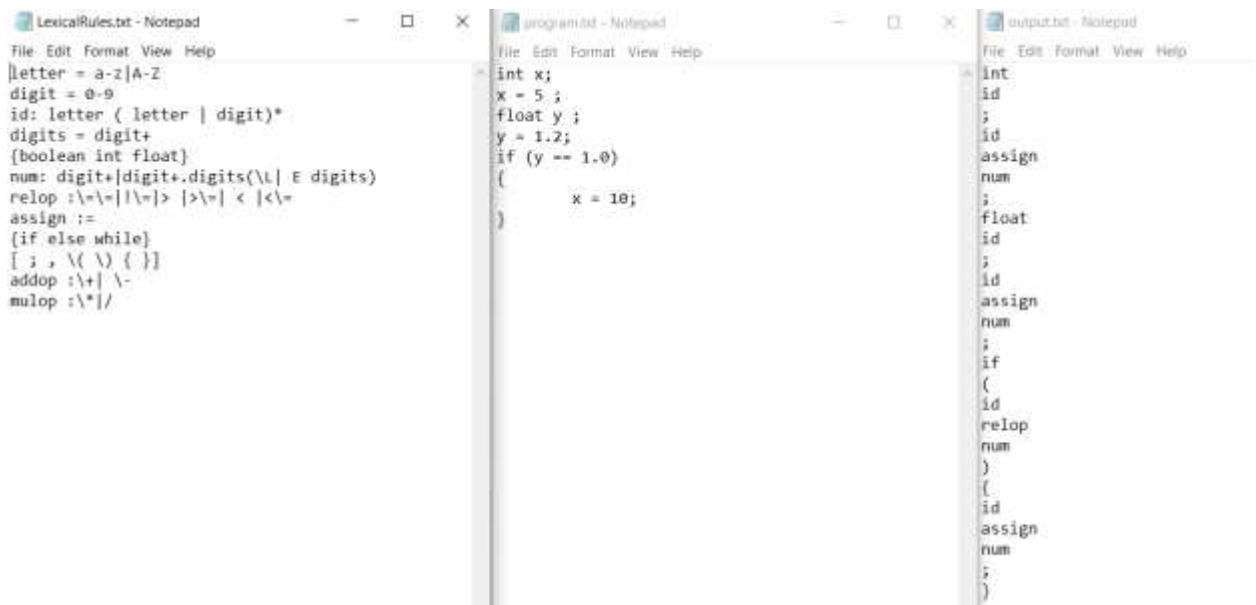
3- If there is an error in the tokens from the last phase, Then the parser simply ignores this token. --Removes it --

4- The terminals are taken from the CFG so when a token is inserted that is not part of the CFG the parser ignores this token. --Removes it—

5- The terminals must be in the form 'id' .

6- Each CFG must begin with # .

## • Sample Runs:



The image shows three Notepad windows side-by-side. The first window, titled 'LexicalRules.txt - Notepad', contains lexical rules for a C-like language, including definitions for letters, digits, identifiers, and various operators. The second window, titled 'program.txt - Notepad', contains a sample C program with variable declarations, assignments, and an if-statement. The third window, titled 'output.txt - Notepad', shows the tokens extracted from the program, such as 'int', 'x', '=', '5', ';', 'float', 'y', '=', '1.2', ';', 'if', '(', 'y', '==', '1.0', '{', 'x', '=', '10', ';', '}', and ')', each on a new line.

```
LexicalRules.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(\.| E digits)
relop :=<|>|<=|>=|<|>|<=|>=
assign :=
{if else while}
[ ; , \{ \} { } ]
addop :=\+| \-
mulop :=\*|/

program.txt - Notepad
File Edit Format View Help
int x;
x = 5 ;
float y ;
y = 1.2;
if (y == 1.0)
{
    x = 10;
}

output.txt - Notepad
File Edit Format View Help
int
id
;
id
assign
num
;
float
id
;
id
assign
num
num
;
if
(
id
relop
num
)
{
id
assign
num
num
;
}
```

Figure: The three text files from phase one.

CFG.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' 'assign' 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 ::= '+' | '-'
```

Figure: the CFG text file.

```
There is a left recursion in # STATEMENT_LIST ::= STATEMENT | STATEMENT_LIST STATEMENT
There is left factoring in # EXPRESSION ::= SIMPLE_EXPRESSION | SIMPLE_EXPRESSION 'relop' SIMPLE_EXPRESSION
There is a left recursion in # SIMPLE_EXPRESSION ::= TERM | SIGN TERM | SIMPLE_EXPRESSION 'addop' TERM
There is a left recursion in # TERM ::= FACTOR | TERM 'mulop' FACTOR
*****The grammar*****
# SIGN ::= '+' | '-'
# FACTOR ::= 'id' | 'num' | '(' EXPRESSION ')'
# TERMDASH ::= 'mulop' FACTOR TERMDASH | ~
# TERM ::= FACTOR TERMDASH
# SIMPLE_EXPRESSIONDASH ::= 'addop' TERM SIMPLE_EXPRESSIONDASH | ~
# SIMPLE_EXPRESSION ::= TERM SIMPLE_EXPRESSIONDASH | SIGN TERM SIMPLE_EXPRESSIONDASH
# EXPRESSIONDASH ::= 'relop' SIMPLE_EXPRESSION | ~
# EXPRESSION ::= SIMPLE_EXPRESSION EXPRESSIONDASH
# ASSIGNMENT ::= 'id' 'assign' EXPRESSION ';'
# WHILE ::= 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
# IF ::= 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
# PRIMITIVE_TYPE ::= 'int' | 'float'
# DECLARATION ::= PRIMITIVE_TYPE 'id' ';'
# STATEMENT ::= DECLARATION | IF | WHILE | ASSIGNMENT
# STATEMENT_LISTDASH ::= STATEMENT STATEMENT_LISTDASH | ~
# STATEMENT_LIST ::= STATEMENT STATEMENT_LISTDASH
# METHOD_BODY ::= STATEMENT_LIST
*****
```

Figure: The grammar in runtime, there was Left recursion & Left factoring.

Note that the new grammar is printed from bottom to top



```

****FIRST****
METHOD_BODY int float if while id
STATEMENT_LIST int float if while id
STATEMENT_LISTDASH int float if while id ~
STATEMENT int float if while id
DECLARATION int float
PRIMITIVE_TYPE int float
IF if
WHILE while
ASSIGNMENT id
EXPRESSION id num ( + -
EXPRESSIONDASH relop ~
SIMPLE_EXPRESSION id num ( + -
SIMPLE_EXPRESSIONDASH addop ~
TERM id num (
TERMDASH mulop ~
FACTOR id num (
SIGN + -
****FOLLOW****
METHOD_BODY $
STATEMENT_LIST $
STATEMENT_LISTDASH $
STATEMENT int float if while id $ }
DECLARATION int float if while id $ }
PRIMITIVE_TYPE id
IF int float if while id $ }
WHILE int float if while id $ }
ASSIGNMENT int float if while id $ }
EXPRESSION );
EXPRESSIONDASH );
SIMPLE_EXPRESSION relop );
SIMPLE_EXPRESSIONDASH relop );
TERM addop relop );
TERMDASH addop relop );
FACTOR mulop addop relop );
SIGN id num (

```

Figure: First & follow. (Note that '~' means epsilon).



The **parsing table** was too big to fit the screen, so we divided it as follow:

```
METHOD_BODY***
id      METHOD_BODY-> STATEMENT_LIST
;
int      METHOD_BODY-> STATEMENT_LIST
float    METHOD_BODY-> STATEMENT_LIST
if        METHOD_BODY-> STATEMENT_LIST
(
-
)
{
-
}
else
while    METHOD_BODY-> STATEMENT_LIST
assign   -
relop    -
addop    -
mulop    -
num      -
+        -
-        -
$        -
```

```
STATEMENT_LIST***
id      STATEMENT_LIST-> STATEMENT STATEMENT_LISTDASH
;
int      STATEMENT_LIST-> STATEMENT STATEMENT_LISTDASH
float    STATEMENT_LIST-> STATEMENT STATEMENT_LISTDASH
if        STATEMENT_LIST-> STATEMENT STATEMENT_LISTDASH
(
-
)
{
-
}
else
while    STATEMENT_LIST-> STATEMENT STATEMENT_LISTDASH
assign   -
relop    -
addop    -
mulop    -
num      -
+        -
-        -
$        -
```

```
STATEMENT_LISTDASH***
id      STATEMENT_LISTDASH-> STATEMENT STATEMENT_LISTDASH
;
int      STATEMENT_LISTDASH-> STATEMENT STATEMENT_LISTDASH
float    STATEMENT_LISTDASH-> STATEMENT STATEMENT_LISTDASH
if        STATEMENT_LISTDASH-> STATEMENT STATEMENT_LISTDASH
(
-
)
{
-
}
else
while    STATEMENT_LISTDASH-> STATEMENT STATEMENT_LISTDASH
assign   -
relop    -
addop    -
mulop    -
num      -
+        -
-        -
$        STATEMENT_LISTDASH->~
```

```
STATEMENT***
id      STATEMENT-> ASSIGNMENT
;
int      STATEMENT-> DECLARATION
float    STATEMENT-> DECLARATION
if        STATEMENT-> IF
(
-
)
{
-
}
else
while    STATEMENT-> WHILE
assign   -
relop    -
addop    -
mulop    -
num      -
+        -
-        -
$        -
```

```

DECLARATION***
id      -
;       -
int     DECLARATION-> PRIMITIVE_TYPE 'id' ';'
float   DECLARATION-> PRIMITIVE_TYPE 'id' ';'
if      -
(       -
)       -
{       -
}       -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     -
+       -
-       -
$       -

PRIMITIVE_TYPE***
id      -
;       -
int     PRIMITIVE_TYPE-> 'int'
float   PRIMITIVE_TYPE-> 'float'
if      -
(       -
)       -
{       -
}       -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     -
+       -
-       -
$       -

ASSIGNMENT***
id      ASSIGNMENT-> 'id' 'assign' EXPRESSION ';'
;       -
int     -
float   -
if      -
(       -
)       -
{       -
}       -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     -
+       -
-       -
$       -

EXPRESSION***
id      EXPRESSION-> SIMPLE_EXPRESSION EXPRESSIONDASH
;       -
int     -
float   -
if      -
(       EXPRESSION-> SIMPLE_EXPRESSION EXPRESSIONDASH
)       -
{       -
}       -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     EXPRESSION-> SIMPLE_EXPRESSION EXPRESSIONDASH
+       EXPRESSION-> SIMPLE_EXPRESSION EXPRESSIONDASH
-       EXPRESSION-> SIMPLE_EXPRESSION EXPRESSIONDASH
$       -

```

```

IF***
id      -
;       -
int     -
float   -
if      IF-> 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
(       -
)       -
{       -
}       -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     -
+       -
-       -
$       -

WHILE***
id      -
;       -
int     -
float   -
if      -
(       -
)       -
{       -
}       -
else    -
while   WHILE-> 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
assign  -
relop   -
addop   -
mulop   -
num     -
+       -
-       -
$       -

```

```

SIMPLE_EXPRESSIONDASH***
id      -
;      SIMPLE_EXPRESSIONDASH->~
int     -
float   -
if      -
(       -
)      SIMPLE_EXPRESSIONDASH->~
{       -
}      -
else    -
while   -
assign  -
relop   SIMPLE_EXPRESSIONDASH->~
addop   SIMPLE_EXPRESSIONDASH-> 'addop' TERM SIMPLE_EXPRESSIONDASH
mulop   -
num     -
+       -
-       -
$       -

TERM***
id      TERM-> FACTOR TERMDASH
;      -
int     -
float   -
if      -
(       TERM-> FACTOR TERMDASH
)      -
{       -
}      -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     TERM-> FACTOR TERMDASH
+       -
-       -
$       -

```

```

SIGN***
id      -
;      -
int     -
float   -
if      -
(       -
)      -
{       -
}      -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     -
+       SIGN-> '+'
-       SIGN-> '-'
$       -

```

```

EXPRESSIONDASH***
id      -
;      EXPRESSIONDASH->~
int     -
float   -
if      -
(       -
)      EXPRESSIONDASH->~
{       -
}      -
else    -
while   -
assign  -
relop   EXPRESSIONDASH-> 'relop' SIMPLE_EXPRESSION
addop   -
mulop   -
num     -
+       -
-       -
$       -

SIMPLE_EXPRESSION***
id      SIMPLE_EXPRESSION-> TERM SIMPLE_EXPRESSIONDASH
;      -
int     -
float   -
if      -
(       SIMPLE_EXPRESSION-> TERM SIMPLE_EXPRESSIONDASH
)      -
{       -
}      -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     SIMPLE_EXPRESSION-> TERM SIMPLE_EXPRESSIONDASH
+       SIMPLE_EXPRESSION-> SIGN TERM SIMPLE_EXPRESSIONDASH
-       SIMPLE_EXPRESSION-> SIGN TERM SIMPLE_EXPRESSIONDASH
$       -

```

```

TERMDASH***
id      -
;       TERMDASH->~
int     -
float   -
if      -
(       -
)       TERMDASH->~
{       -
}       -
else    -
while   -
assign  -
relop   TERMDASH->~
addop   TERMDASH->~
mulop   TERMDASH-> 'mulop' FACTOR TERMDASH
num     -
+       -
-       -
$       -

FACTOR***
id      FACTOR-> 'id'
;       -
int     -
float   -
if      -
(       FACTOR-> '(' EXPRESSION ')'
)       -
{       -
}       -
else    -
while   -
assign  -
relop   -
addop   -
mulop   -
num     FACTOR-> 'num'
+       -
-       -
$       -

```

Each Nonterminal with its entry in the parse table is printed alone. The '-' sign means that nothing is present when this terminal is the input and '~' means epsilon.



-Another Sample Run with different program.txt:

program.txt - Notepad	output.txt - Notepad
File Edit Format View Help	File Edit Format View Help
float x;	float
x=1.2;	id
while(x>6.0)	;
{	id
x=2.8;	assign
}	num
	;
	while
	(
	id
	relop
	num
	)
	{
	id
	assign
	num
	;
	}

Figure: The program txt file and the output –Tokens –

```

outputPhaseTwo.txt - Notepad
File Edit Format View Help
[STATEMENT_LIST] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id, float]
[STATEMENT_LISTOASH, STATEMENT] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id, float]
[STATEMENT_LISTOASH, DECLARATION] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id, float]
[STATEMENT_LISTOASH, 'id', PRIMITIVE_TYPE] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id, float]
[STATEMENT_LISTOASH, 'id', 'float'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id, float]
[STATEMENT_LISTOASH, 'id'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id]
[STATEMENT_LISTOASH, 'id'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id]
[STATEMENT_LISTOASH, 'id'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id]
[STATEMENT_LISTOASH, 'id'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id, ;, id]
[STATEMENT_LISTOASH, STATEMENT] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id]
[STATEMENT_LISTOASH, ASSIGNMENT] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id]
[STATEMENT_LISTOASH, 'id', EXPRESSION, 'assign', 'id'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign, id]
[STATEMENT_LISTOASH, 'id', EXPRESSION, 'assign'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num, assign]
[STATEMENT_LISTOASH, 'id', EXPRESSION] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num]
[STATEMENT_LISTOASH, 'id', EXPRESSIONDASH, SIMPLE_EXPRESSION] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num]
[STATEMENT_LISTOASH, 'id', EXPRESSIONDASH, SIMPLE_EXPRESSIONDASH, TERM] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num]
[STATEMENT_LISTOASH, 'id', EXPRESSIONDASH, SIMPLE_EXPRESSIONDASH, TERMDASH, FACTOR] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num]
[STATEMENT_LISTOASH, 'id', EXPRESSIONDASH, SIMPLE_EXPRESSIONDASH, TERMDASH, 'num'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;, num]
[STATEMENT_LISTOASH, 'id', EXPRESSIONDASH, SIMPLE_EXPRESSIONDASH, TERMDASH] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;]
[STATEMENT_LISTOASH, 'id', EXPRESSIONDASH, SIMPLE_EXPRESSIONDASH] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;]
[STATEMENT_LISTOASH, 'id', EXPRESSIONDASH] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;]
[STATEMENT_LISTOASH, 'id'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;]
[STATEMENT_LISTOASH, 'id'] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;]
[STATEMENT_LISTOASH, STATEMENT] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;]
[STATEMENT_LISTOASH, WHILE] [$, ), ;, num, assign, id, (, ), num, relop, id, (, while, ;]

```



