Phase One
Lexical Analyzer Generator

Mayar El Mahdy – 4639.

El Zahraa Emara – 4558.

Part One:

Data Structures used:

1-Stack: Used this data structure in order to help in implementing the NFA automata, by pushing the characters of the Regular expression and popping them when I find either of these cases:

- When the character read is '(' , It will pop the elements from the stack
- When the expression is all pushed in the stack, then pop the expression and start building the nodes.
- When the character is ')' then pop the elements from the stack.

```
for(int i=0;i<expression.length();i++)
{
    push = 1;
    if(expression.charAt(i)=='(') {
        openB=1;
        if(OR>=1)
            push=1;
        else {
        begin = popStack(begin,OR,0);
        OR = 0;
        push = 0;
        }

    }
    else if(expression.charAt(i)==')') {

        //pop till you find the closed bracket
        if(OR > 1)
            push=1;
        else
        {
        begin = popStack(begin,OR,1);
        OR=0;
        openB=0;
        }
    }
     if(expression.charAt(i) == '|')
        {
            push = 1;
            OR++;
        }
     if(push == 1) {

         stack.push(expression.charAt(i));
     }
}
if(!stack.empty())
    begin = popStack(begin,OR,0);
```

Figure : Usage of Stack

2- ArrayList : Used the array list to add strings into a list , this was used several times in the code like in the class LexicalRules when reading the input file I simple add the Line I read into an ArrayList to handle it.

3- 2D arrays: This helps in storing the transition table.

4- Queue: Used to Build the DFA, whenever a new state appears it is inserted to the queue of states to be checked then added to the transition table of the DFA.

Part two:
Algorithms & Techniques used:

1- Implemented an array of Nodes that was used in building the automata , It is a **graph** but implemented from scratch in order to add more functions to the nodes present.

2- The 2D array that was mentioned in the data structures section.

3- Split() , used this algorithm to split when I see the occurrence of a certain String

For example :

```
for(int i = 0 ; i<regDef.size();i++)
{
    //represent each regular definition with a symbol
    //which is a letter from it , make it Capital letter
    //we need to substitute the regular expression with this symbol
    String temp = regDef.get(i);
    String[] temp2 = temp.split("=");
    regDefinition.name(temp2[0] , temp2[1]);
}
regDefinition.endNames();
nfa = new NFA(regDefinition);
for(int i=0;i<regExp.size();i++)
{
    //split the expression when you see ":"
    // the name of this expression LHS will be the name of it's NODE --FINISH STATE--
    String temp = regExp.get(i);
    String[] temp2 = temp.split(":");
    //build the NFA for this expression
    if(temp2.length>2)
    {
        for(int j=2;j<temp2.length;j++) {

            temp2[1] = temp2[1]  + ':' + temp2[j];
        }
    }

    temp2[1]=regDefinition.contain(temp2[1]);
    nfa.buildNFA(temp2);
}
```

Figure: Split()

Here I split the regular expression and the regular definition whenever I find (:) , (=).

4-replace() , Used this algorithm to replace a certain String with another one .

For example:

```
public String contain(String exp)
{
    if(exp.contains("E"))
        exp = exp.replace("E", "^");
    for(int i=actualNames.size()-1;i>-1;i--)
    {
        if(exp.contains(actualNames.get(i)))
        {

            exp = exp.replace(actualNames.get(i), (names.get(i)));

        }
    }
    //we also change any thing that has (+) to (character)(character
    if(exp.contains("+") && !exp.contains("\\+"))
    {

        exp = removePlus(exp);
    }
    //remove the exponent to be (^)

    return exp;
}
```

Figure: Usage of replace()

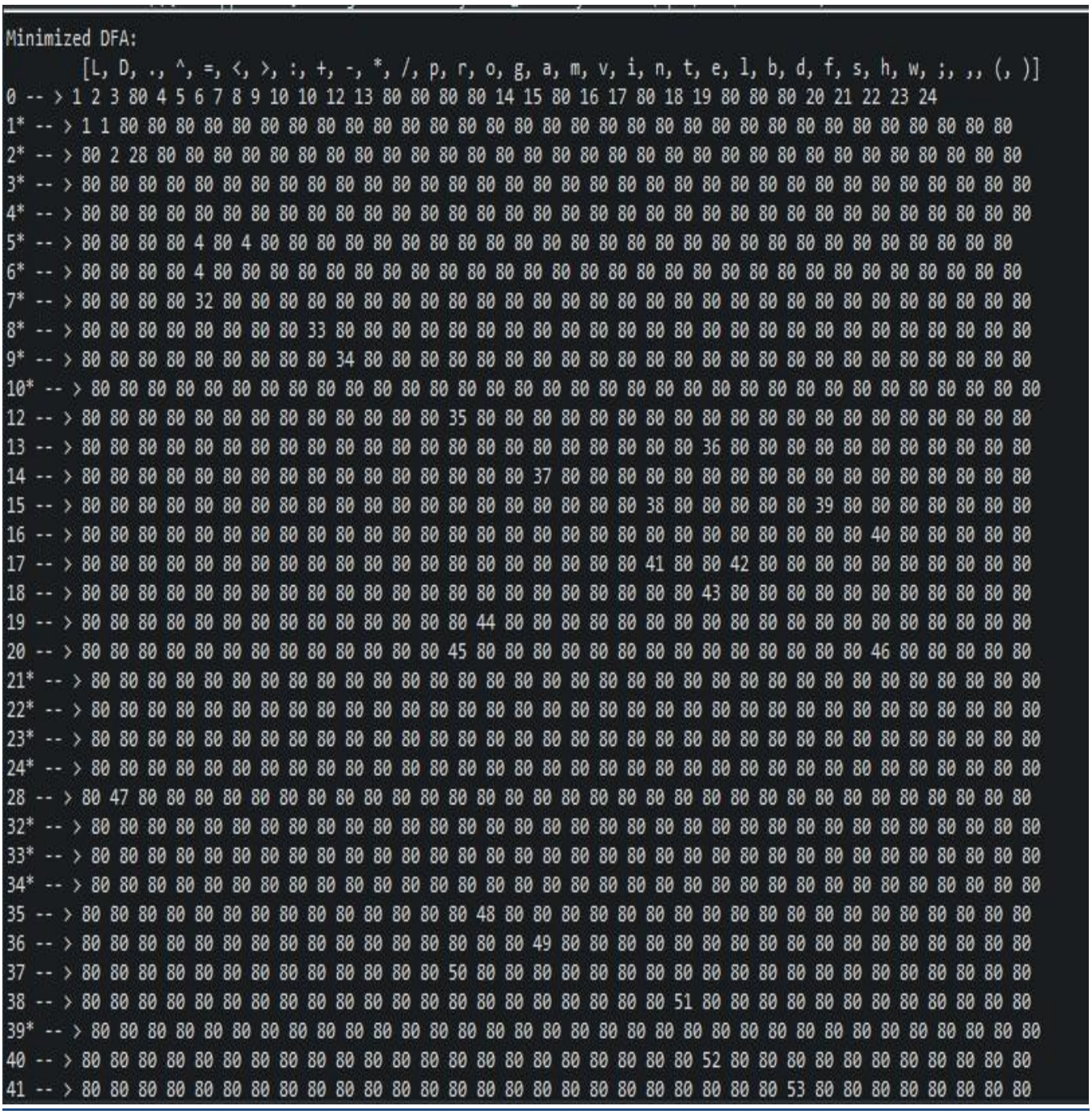
In the figure above, each regular expression containing a reference to a regular definition will be replaced by its alternative, also E will be (^).

5-Equivalence Algorithm for DFA minimization:
First, we get the zero equivalence of the states by dividing them into two sets, one for the accepting states & the other one for the non-accepting ones.
Then, we get the 1 equivalence by taking the first set and looping on all of its members, if equivalent they stay in the same set, else we create a new set and add the non-equivalent state to it.
The last step is repeated recursively for N times until the N and N-1 equivalences are identical.
*Equivalence between two states is achieved if and only if they have the same language name and each two corresponding outputs of the two states belong to the same set from the previous equivalence.

Part three:

Transition Table for minimal DFA:

80 is the Ø state in this example.



```
Minimized DFA:
        [L, D, ., ^, =, <, >, :, +, -, *, /, p, r, o, g, a, m, v, i, n, t, e, l, b, d, f, s, h, w, ;, ,, (, )]
0 -- > 1 2 3 80 4 5 6 7 8 9 10 10 12 13 80 80 80 80 14 15 80 16 17 80 18 19 80 80 80 20 21 22 23 24
1* -- > 1 1 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
2* -- > 80 2 28 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
3* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
4* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
5* -- > 80 80 80 80 4 80 4 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
6* -- > 80 80 80 80 4 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
7* -- > 80 80 80 80 32 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
8* -- > 80 80 80 80 80 80 80 80 33 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
9* -- > 80 80 80 80 80 80 80 80 80 34 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
10* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
12 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 35 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
13 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 36 80 80 80 80 80 80 80 80 80 80
14 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 37 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
15 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 38 80 80 80 80 80 39 80 80 80 80 80 80
16 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 40 80 80 80 80 80
17 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 41 80 80 42 80 80 80 80 80 80 80 80 80
18 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 43 80 80 80 80 80 80 80 80 80 80 80
19 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 44 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
20 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 45 80 80 80 80 80 80 80 80 80 80 80 80 80 80 46 80 80 80 80 80
21* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
22* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
23* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
24* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
28 -- > 80 47 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
32* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
33* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
34* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
35 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 48 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
36 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 49 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
37 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 50 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
38 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 51 80 80 80 80 80 80 80 80 80 80 80
39* -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
40 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 52 80 80 80 80 80 80 80 80 80 80
41 -- > 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 53 80 80 80 80 80 80 80 80
```

Part Four:

Test files outputs:

1-Test Case Output 1

Test Case 1	Test Case 2	Test Case 3	Test from pdf
program	program	program	int
id	id	id	id
;	;	;	,
var	var	var	id
id	id	id	,
,	,	,	id
id	id	id	,
:	:	:	id
integer	integer	integer	;
;	;	;	while
begin	begin	begin	(
id	id	id	id
assign	assign	incop	relop
num	num	;	num
;	;	id	)
while	id	decop	{
id	assign	;	id
relop	floatNum	while	assign
num	;	id	id
do	id	relop	addop
begin	assign	num	num
id	num	do	;
assign	;	begin	}
id	while	id	
addop	id	assign	
num	relop	id	
;	num	addop	
read	do	num	
(	begin	;	
id	id	read	
)	assign	(	
;	id	id	
if	addop	)	
id	num	;	
relop	;	if	
num	read	id	
then	(	relop	
id	id	num	

assign id addop num else id assign id addop id end ; write (id , id) end .	) ; if id relop num then id assign id addop floatNum else id assign id addop id end ; write (id , id) end .	then id assign id addop floatNum else id assign id addop id end ; write (id , id) end .	

Part Five:

Assumptions:

- Assumed that the epsilon has a symbol (~)
- Assumed that whenever a (E) was found in the regular expression then it will be replaced with the symbol (^) as the (E) means exponent.
- Assumed that the regular definitions will be replaced with the capital letter of its first letter **ex**: letter = A-Z|a-z Would become : L = A-Z|a-z

  So that it will be replaced when it is found in any regular expression **ex:** id:letter* would become id:L*

- Assumed that the arrows would carry a value that is a character **always** , so when I read the regular expression if there are two character they would be separated as two different nodes
- The regular definitions will be replaced with an exception in every node
  So that wouldn't be any confusion if a node has several arrows
  Making the regular expression L-{any other character that has an arrow in this node}.
- The code is sensitive to spaces for example in lexical rules.txt the punctuations should be separated by spaces
  [\(\(]] → There should be a space between them.