



# Programming Language Translation Project Report

---

## Java Compiler

Mayar El Mahdy – 4639.

El Zahraa Emara – 4558.

## Contents

Phase one .....	3
Phase Two .....	11
Phase Three .....	30

# Phase One

## Lexical Analyzer Generator

---

## 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.

```

//push = 1;
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 1 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.

## 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 2 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 3 replace() demonstration

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

### Transition Table for minimal DFA:

[illegible]

## Part Four: Test files outputs:

### 1-Test Case Output 1

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



<pre>else id assign id addop id end ; write ( id , id ) end .</pre>	<pre><u>relop</u> <u>num</u> then id assign id <u>addop</u> floatNum else id assign id <u>addop</u> id end ; write ( id , id ) end .</pre>	<pre><u>addop</u> floatNum else id assign id <u>addop</u> id end ; write ( id , id ) end .</pre>	
---	--	--	--

## Part Five: Assumptions:

- Assumed that the epsilon has a symbol ( $\sim$ )
- 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.

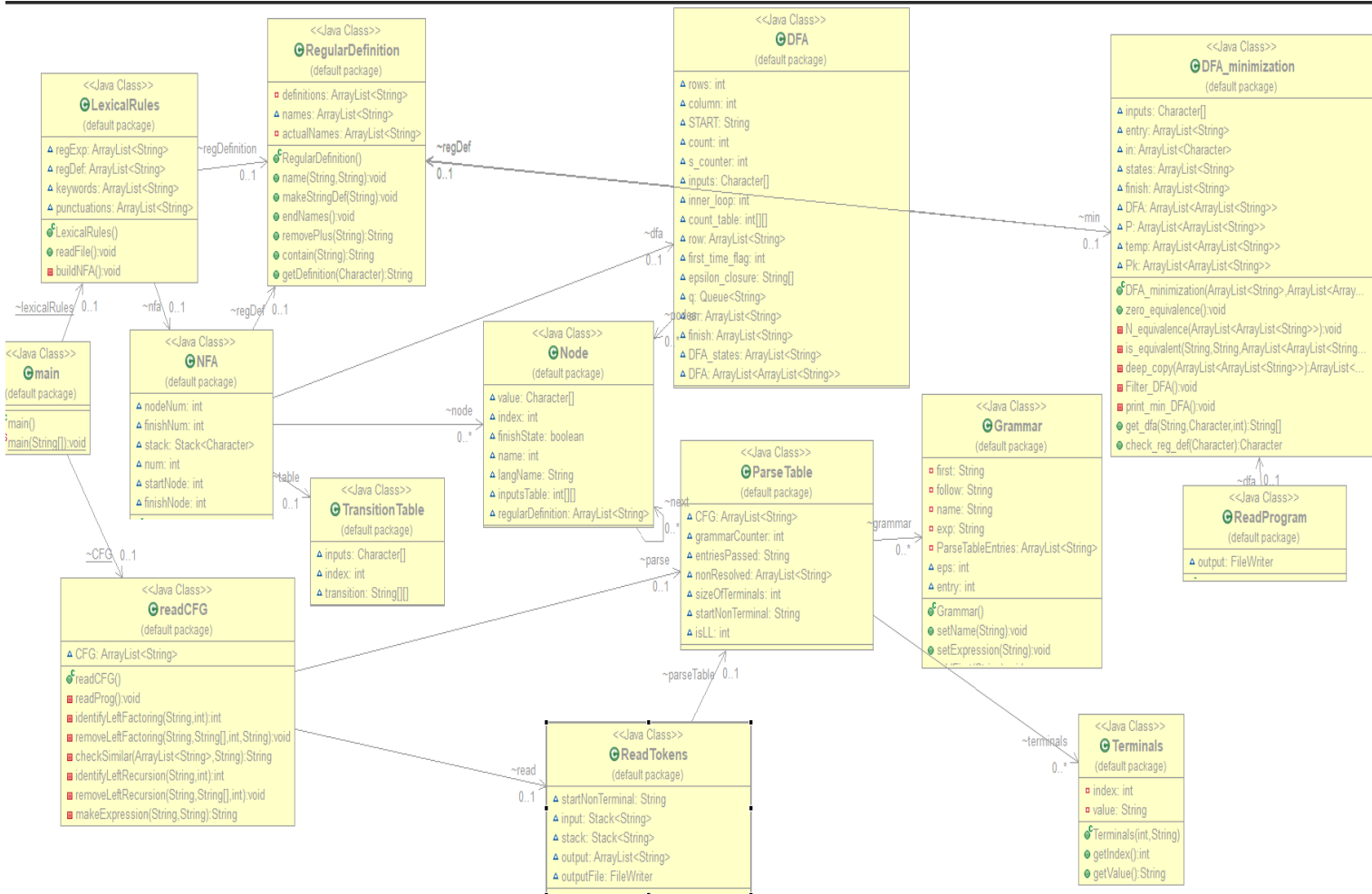
# Phase Two

## Parser Generator

---

## Class UML 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 (~) 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 4 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 its name then loop :

- ❖ If there comes a terminal after its 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 its 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("~")) { //if there is an epsilon then remove this non ter
            first = first.replace("~", "");
            //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 5 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;
}
private void removeLeftFactoring(String name, String[] exp,int j, St

```

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/non-terminals 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 exp
    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 exp
            exp[i] = exp[i].replaceFirst(similar, " ");

            if(exp[i].equals(" "))
                eps=1;
            else
                newExp = newExp + "|" + exp[i];
        }
        else {
            //add to the old exp
            temp = temp + "|" + exp[i];
        }
    }

    temp = temp + "|" + " "+similar + name2 ;
    temp = temp.replaceAll("\\s+", " ");
}

```

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.

```

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);
    }
}
}
}

```

## -Left Recursion:

### 1-Identify it:

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 .

```
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;
}
```

### 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;
                    }
                }
            }
        }
    }
}
```

```
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);
}
```

### **-Grammar Class:**

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

**1-initializeParseTableEntries:** This is used to fill up the ArrayList that will hold the entries to the parse table, so it initializes 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 its 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);
```

Figure 6 Stack method

## 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.

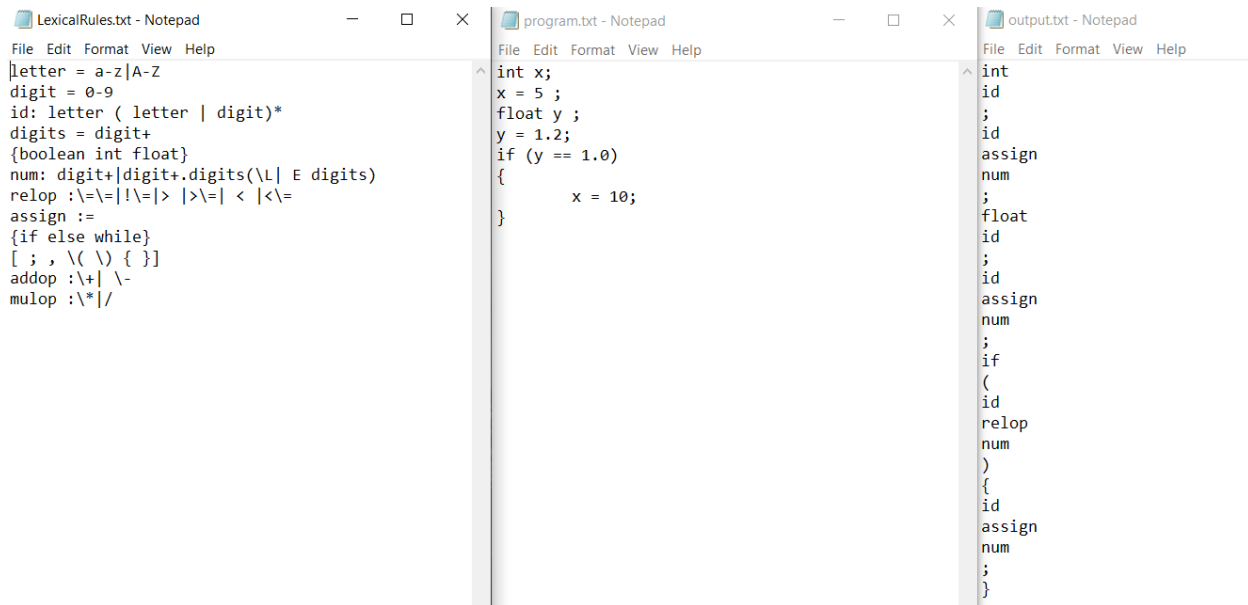
3- If there is an error in the tokens from the last phase, Then the parser simply ignores this token. --Removes it --

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, 'LexicalRules.txt', contains lexical rules for identifiers, digits, and operators. The second window, 'program.txt', contains a small C-like program snippet. The third window, 'output.txt', shows the output of the lexical analysis, listing tokens like 'int', 'id', 'float', 'num', and operators.

```

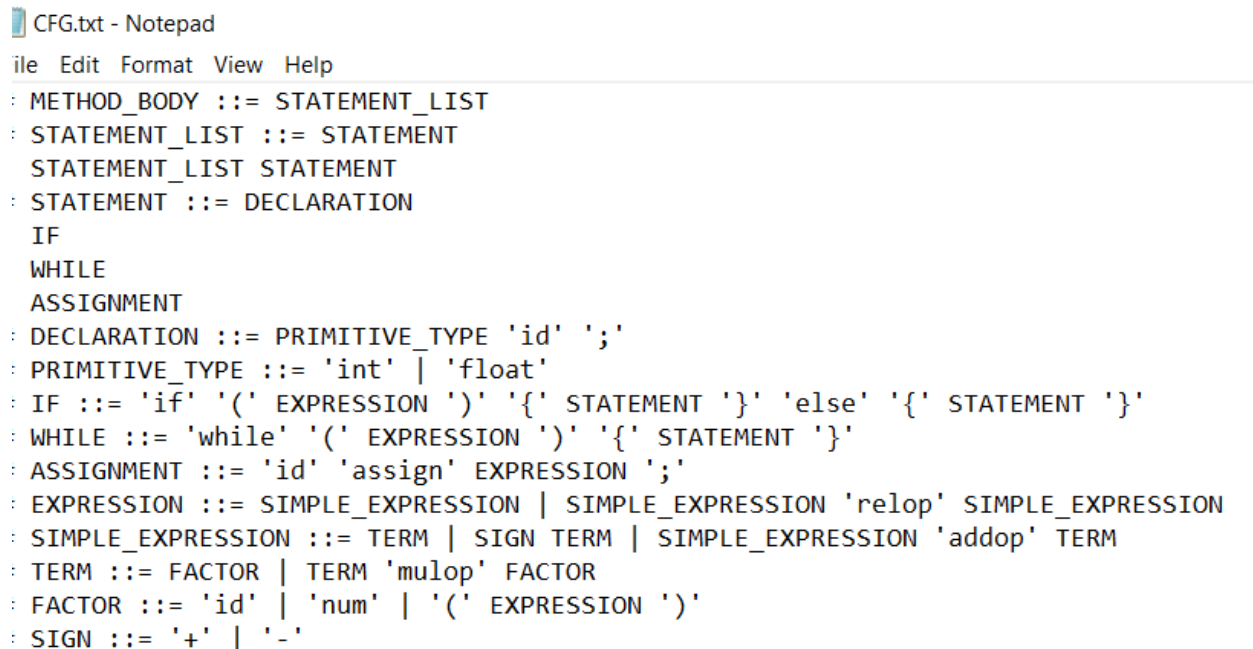
LexicalRules.txt - Notepad
File Edit Format View Help
letter = a-z|A-Z
digit = 0-9
id: letter ( letter | digit)*
digits = digit+
{boolean int float}
num: digit+|digit+.digits(\. E digits)
relop := \=|!=|>|>|=|<|<|=
assign :=
{if else while}
[ ; , \ ( \ ) { } ]
addop := \+| \-
mulop := \*|/

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

```

Figure 7 The three text files from phase one.



The image shows a Notepad window titled 'CFG.txt' containing a Context-Free Grammar (CFG) for the language. The rules define the structure of statements, expressions, and terms.

```

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 8 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
*****END*****

```

Figure 9 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 10 First & follow. (Note that '~' means epsilon)



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

METHOD_BODY***			STATEMENT_LISTDASH***		
id	METHOD_BODY->	STATEMENT_LIST	id	STATEMENT_LISTDASH->	STATEMENT STATEMENT_LISTDASH
;	—		;	—	
int	METHOD_BODY->	STATEMENT_LIST	int	STATEMENT_LISTDASH->	STATEMENT STATEMENT_LISTDASH
float	METHOD_BODY->	STATEMENT_LIST	float	STATEMENT_LISTDASH->	STATEMENT STATEMENT_LISTDASH
if	METHOD_BODY->	STATEMENT_LIST	if	STATEMENT_LISTDASH->	STATEMENT STATEMENT_LISTDASH
(	—		(	—	
)	—		)	—	
{	—		{	—	
}	—		}	—	
else	—		else	—	
while	METHOD_BODY->	STATEMENT_LIST	while	STATEMENT_LISTDASH->	STATEMENT STATEMENT_LISTDASH
assign	—		assign	—	
relop	—		relop	—	
addop	—		addop	—	
mulop	—		mulop	—	
num	—		num	—	
+	—		+	—	
-	—		-	—	
\$	—		\$	STATEMENT_LISTDASH->~	
STATEMENT_LIST***			STATEMENT***		
id	STATEMENT_LIST->	STATEMENT STATEMENT_LISTDASH	id	STATEMENT->	ASSIGNMENT
;	—		;	—	
int	STATEMENT_LIST->	STATEMENT STATEMENT_LISTDASH	int	STATEMENT->	DECLARATION
float	STATEMENT_LIST->	STATEMENT STATEMENT_LISTDASH	float	STATEMENT->	DECLARATION
if	STATEMENT_LIST->	STATEMENT STATEMENT_LISTDASH	if	STATEMENT->	IF
(	—		(	—	
)	—		)	—	
{	—		{	—	
}	—		}	—	
else	—		else	—	
while	STATEMENT_LIST->	STATEMENT STATEMENT_LISTDASH	while	STATEMENT->	WHILE
assign	—		assign	—	
relop	—		relop	—	
addop	—		addop	—	
mulop	—		mulop	—	
num	—		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 it goes to nothing when this terminal is the input and '~' means epsilon.



# Phase three

## Java Byte Code Generation

---

## Description:

We implemented this phase using our own parser generator implemented in phase two, instead of bison –Bonus Part –

This phase won't run unless phase two has no errors i.e.: No semantic errors in the program.

## Functions explained:

1) **handleConstants** – This function handles the cases of declaration of a new variable, applying operation on it

```
private void handleConstants(String program,int newVar) throws IOException {
//generate byte code for int , float with/without operations on them
char first = program.replaceAll("\\s","").charAt(0);

if(!program.contains("="))
{
    variableDeclaration.add(first);
    //if the variable doesn't have a declaration ie: int x ; then default is to put x=0
    String write = line + ":" + " " + first + "const_0" + "\n" ;
    line++;

    if(numOfVariables <= 3) {
        write = write + + line + ":\t" + first+"store_"+numOfVariables;
        line++;
    }
    else
    {
        write = write + + line + ":\t" +first+"store "+numOfVariables;
        line+=2;
    }

    if(dontWrite == 0 && isWhile == 0)
        writeByteCode(write);
    else if(isWhile == 0)
        writeTemp = writeTemp + "\n" + write ;
    else
        tempWhile = tempWhile + "\n" + write;
    variable.add(program.split(" ")[1].replace(";", ""));
    numOfVariables++;

    return;
}
```

The above handles the case of int x; or float x;

We set the default of any new variable as (0) –Initialize the variable- so that there wont be any problems faced if we don't initialize the variable

First we check if the line contains (=) – no initialization – then we add it's type whether it is an int or float to an arrayList called **variableDeclaration** and also add the new variable to another arrayList holding the names of each variable called **variable**

The variable **numOfVariables** is just a counter to count the number of variables found in the program, this helps in writing the bytecode as when storing, loading the variables they should be placed in certain places.

The variable **line** is also a counter, but it counts the bytes that is used by bytecode.

In bytecode there are default store mnemonics either in int(i) or float(f)

Store\_0 Store\_1 Store\_2 Store\_3 -> Take only 1 byte

Store 4~more -> Take 2 bytes, one byte for opcode one byte for the number, as it is not built in

That's why the code checks if numOfVariables is <=3 to see which mnemonic to use.

dontWrite & isWhile are both variables used when (if) or (while) is present, they will be explained later.

```
private void writeByteCode(String write) throws IOException
{
    fileWriter.write(write + "\n");
}
```

If there is no (if) nor (while) then we write directly to the output file .



## 2) Primitive types with declaration:

```
String[] split = program.split("=");
String check = split[1].replace("\\s", "").replace(";", "");

int temp = numOfVariables;

if(newVar == 1)
    variableDeclaration.add(first);
else {
    System.out.println(split[0].replaceAll("\\s", ""));
    first = variableDeclaration.get(findVariables(split[0].replaceAll("\\s", "")));
    numOfVariables = findVariables(split[0].replaceAll("\\s", ""));
}

try {
    Float tryIt = Float.parseFloat(check);
    handleNum(check, first, tryIt, 1);
    numOfVariables = temp;
    if(newVar==1) {
        variable.add(split[0].split(" ")[1]);
        numOfVariables++;
    }
} catch(NumberFormatException nfe)
{
    handleOp(split[1].replace(";", ""), first);
    numOfVariables = temp;
    if(newVar==1) {
        variable.add(split[0].split(" ")[1]);
        numOfVariables++;
    }
}
```

The rest of `handleConstant`—function, checks if the declaration of a certain constant is an operation or a number

It first splits on `(=)` then `String check` is the number/Operation after the `(=)`

The `int` called `newVar` is used as a flag to check if this variable was already declared or not, if not then we have to get its primitive type (`int` or `float`) and its index.—To load and store it—

The **try catch** is used to check if the declaration is a number or an operation

## b-1) handleNum

There are certain cases for a number to be loaded to the stack in bytecode, here it shows if it is a negative number.

```
private void handleNum(String check,char first,float num , int storeIt) throws IOException
{
    String write;
    if(check.contains("-"))
    {
        num = num * -1;
        if(num == 1)
        {
            write = line + ":" + "\t"+first+ "const_" + check.replace("-", "m") ;
            line++;
            if(storeIt ==1 ) {
                if(numOfVariables <= 3) {
                    write = write + "\n" + line + ":\t" +first+ "store_"+numOfVariables;
                    line++;
                }
                else
                {
                    write = write + "\n" + line + ":\t" + first+"store "+numOfVariables;
                    line+=2;
                }
            }
            if(dontWrite == 0 && isWhile == 0)
                writeByteCode(write);
            else if(isWhile == 0)
                writeTemp = writeTemp + "\n" + write ;
            else
            {
                tempWhile = tempWhile + "\n" + write;
            }
            return;
        }
    }
}
```

If num == -1 then the bytecode mnemonic is iconst\_m1

Then we check if we want to store it or not yet , if yes we do the same as we did before and that is checking the numOfVariables and choosing the right mnemonic to use and how many bytes .

And check the (dontWrite and isWhile) that will be explained later.

```

    if(num > 5 ) {
        if(first == 'i') {
            char length = 'b';
            if(num > 127)
                length = 's';
            write = line + ":" + length + "ipush\t" + check ;
            if(length == 's')
                line+=3;
            else
                line+=2;
            if(storeIt == 1) {
                if(numOfVariables <= 3) {
                    write = write + "\n" + line + ":\t" + "istore_" + numOfVariables;
                    line++;
                }
                else
                {
                    write = write + "\n" + line + ":\t" + "istore " + numOfVariables;
                    line+=2;
                }
            }
        }
    }

```

If a number is greater than 5 then we need to use different mnemonics for int and float, the above shows the case of int

If the num is short, meaning the num > 127 then we will use sipush num, this takes 3 bytes, 1 for opcode 2 for the num

If it is less then, it is a byte → bipush num, this takes 2 bytes, 1 for opcode 1 for num.

```

    }
    else
    {
        write = line + ":" + "\t" + "ldc " + check ;
        line+=2;
        if(storeIt ==1) {
            if(numOfVariables <= 3) {
                write = write + "\n" + line + ":\t" + "fstore_" + numOfVariables;
                line++;
            }
            else
            {
                write = write + "\n" + line + ":\t" + "fstore " + numOfVariables;
                line+=2;
            }
        }
        if(dontWrite == 0 && isWhile == 0)
            writeByteCode(write);
        else if(isWhile == 0)
            writeTemp = writeTemp + "\n" + write ;
        else
            tempWhile = tempWhile + "\n" + write;
        return;
    }
}

```

This is the same as the previous but if the declaration is float we use → ldc num with 2 bytes

## b-2) handle\_A\_op:

This function is called only when an arithmetic expression is on the RHS of the assign operator.

For a parameter, it takes the expression in the postfix form.

Ex:  $\text{int } z = 2 * (2 + x) + 7 \rightarrow$   
 $\text{int } z = 2 \ 2 \ x \ + \ * \ 7 \ +$

**Note:** elements MUST be separated by spaces.

First, we store the elements of the postfix expression in an array.  
 Then we loop on the array entries.

- 1) if the element is an arithmetic operator (+, -, \*, /, %) we print the suitable bytecode for it for example: iadd, fsub, imul .. etc.
- 2) if the element is a digit, pass it to the method (handleNum) which will also print its bytecode ex: iconst\_2
- 3) Lastly, if the element is a variable, we send it to the method (numOrVariable) to get its index in the symbol table, then print the bytecode.  
 ex: iload\_1

Now, after we processed the postfix expression we need to know where to store it.

we have two possibilities; either store it in a previously declared variable or store it in a new variable  
 ex:  $x = 2+1$  or  $\text{int } x = 2+1$

```
private void handle_A_op(String post,
    char first, String program) throws IOException {
    System.out.println(post);
    String[] postfix = post.split(" ");
    String write = "";

    for (int i = 0; i < postfix.length; i++) {
        String t = postfix[i];
        if (t.equals("+")) {
            write = "\n" + line + ": " + first + "add";
            if(isBoolean)
                tempBoo = tempBoo + write;
            else if (isWhile == 0) {
                writeTemp = writeTemp + "\n" + write;
            } else
                tempWhile = tempWhile + "\n" + write;
            line++;
        } else if (t.equals("-")) {
            write = "\n" + line + ": " + first + "sub";
```

```
        else {
            // either digit or variable
            if (Character.isDigit(t.charAt(0))) {

                Float tryIt = Float.parseFloat(t);
                handleNum(t, first, tryIt, 0);
            }
            else if (Character.isAlphabetic(t.charAt(0))) {

                int num_one = numOrVariable(t, first); // get index of
```

```
String[] LHS = program.split("="); // int x || x
if(program.contains("int") || program.contains("float"))
    //undeclared variable
{
    String[] v = LHS[0].split(" "); // x
    String var = v[1].trim();
    write = "\n" + line + ": " + first
        + "store " + numOfVariables;
    if(isBoolean)
        tempBoo = tempBoo+write;
    else if (isWhile == 0) {
        writeTemp = writeTemp + "\n" + write;
    } else
        tempWhile = tempWhile + "\n" + write;
    line++;
    variable.add(var); // Declare the new variable
    numOfVariables++;
}
```

If the variable was declared before we send it to the method (numOrVariable) to get its index, then print the bytecode. Ex: istore\_3. Else we declare it then repeat the same steps.

### 3) If Condition, else:

There are two variables used

1) Int **dontWrite** 2) String **writeTemp**

**dontWrite** will be set to 1 when the program starts an if condition and is set to 0 when the if conditions finishes (when it finds '}' )

**writeTemp** is used to store the bytecode of the if till it finishes , this is used to ensure back tracing where we set the number that should point to the end of the if with (~) then replace it when we finish reading the if condition.

If there is an **else** the bytecode go to ~ also has the (~) symbol till we reach the end of this condition.

The following figure shows the function that handles the if , in bytecode the standard if is used in two ways; either to compare with zero Or compare with number/variable.

```
private void ifCondition(String program ) throws IOException
{
    String[] split = program.split("\\(");
    String condition = split[1].replace(")", "");
    String op2 = getOpCondition(condition);
    //check if the condition has ZERO or not
    String op1;
    whileNum1 = line;
    if(condition.contains("0"))
        op1 = "if";

    else
        op1 = "if_icmp";
    //load the variables and numbers that will be compared
    int var1 = findVariables(condition.split("\\(" + temp)[0].replaceAll("\\s", ""));

    char first = variableDeclaration.get(var1);
    int var2 = -1;
    if(!condition.split("\\(" + temp)[1].equals("0"))//if it is zero dont load it as the default
        var2 = numOrVariable(condition.split("\\(" + temp)[1] , first);

    if(var1 != -1) {
        if(isWhile == 0) {
            writeByteCode(line+ " : " + first+"load_"+var1);
            line++;
        }
        else {
            if(var1 <= 3){
                tempWhile = tempWhile + "\n" + line+ " : " + first+"load_"+var1 ;
                line++;
            }
            else {
                tempWhile = tempWhile + "\n" + line+ " : " + first+"load "+var1 ;
                line+=2;
            }
        }
    }
}
```

```

}
private String getOpCondition(String condition)
{
    //check the condition and return the ending
    if(condition.contains("!=")) {
        temp = "!=";
        return "eq";
    }
    if(condition.contains("==")) {
        temp = "==";
        return "ne";
    }
    if(condition.contains("<=")) {
        temp = "<=";
        return "gt";
    }
    if(condition.contains(">=")) {
        temp = ">=";
        return "lt";
    }
    if(condition.contains("<")) {
        temp = "<";
        return "ge";
    }
    if(condition.contains(">")) {
        temp = ">";
        return "le";
    }
    return "";
}

```

```

if(tempString.contains("")) )
{
    if(dontWrite == 1) {

        int t = line;

        tempString = buffer.readLine();
        if(tempString != null )
        {

            if(tempString.contains("else"))
                t = line +3;

        }
        dontWrite = 0;
        if(isWhile == 0) {
            writeTemp = writeTemp.replace("~", Integer.toString(t));
            writeByteCode(writeTemp);
        }
        else
            tempWhile = tempWhile.replace("~",Integer.toString(t));

        writeTemp = "";

    }
}

```

The bytecode mnemonics are similar but work differently, So the code first checks if the condition is compared to a zero then the (if) is the start of the mnemonic else the start is (if\_icmp).

Then var1 is always a variable so we get the variable's index and load it, then if we compare with something other than zero so we need to get the variable or number as we did in previously.

This gets the ending of the bytecode mnemonic

Ex: ifeq, if\_icmp, ifne, if\_icmpne

```

}
if(isWhile == 1 && tempString.contains(""))
{

    ifCondition(whileCondition);
    tempWhile = tempWhile.replace("^", Integer.toString(whileNum1));
    isWhile = whileNum1 = whileNum2 = 0;
    writeByteCode(tempWhile);
    tempWhile = "";
}

if(tempString == null)
    break;

```

```

private void ifCondition(String program ) throws IOException
{

    String[] split = program.split("\\(");

    String condition = split[1].replace(")", "");
    String op2 = getOpCondition(condition);
    //check if the condition has ZERO or not
    String op1;
    whileNum1 = line;
}

```

## Handling the back patching in if condition:

When dontWrite is equal to 1 it means that there was an if condition or else so we need to check first the next line to be read if there is an (else) so the (~) should be replaced by the line number after the (go to ) statement

Which is:  $t = \text{line} + 3$

Else it is replaced by the current line that will be written.

### 3)While:

Three variables are used for the while:

**String tempWhile;** → used to store the bytecode that is inside the while loop till it reaches the ( } ) character.

**Int isWhile;** → used as a flag to help in writing the bytecode in the **tempWhile** string other than the file.

**Int whileNum1, whileNum2;** → Two numbers are used in the back tracing both will be substituted at the end.

### Back patching in while:

The int whileNum1 is going to be the line before writing the if condition of the while loop i.e.: in bytecode the while loop's bytecode ends with its condition.

Ex:

While( $x \neq 0$ ) → the ( $x \neq 0$ ) will be considered an if statement and the number it follows is the **beginning** of the while loop.—whileNum2 --

```

    }
    if(isWhile == 1 ) {
        if(!split[0].replaceAll("\\s", "").equals("if"))
            tempWhile = tempWhile + "\n" + line + ":" + "\t" + op1 + op2 + " " + whileNum2;
        else {
            dontWrite = 1;
            tempWhile = tempWhile + "\n" + line + ":" + "\t" + op1 + op2 + " ~";
        }
        line+=3;
        return;
    }

```

```

    }
    private void whileCond(String program)
    {
        whileCondition = program;

        isWhile = 1;
        tempWhile = line + ":\t" + "go to" + "\t^" ;
        line+=3;
        whileNum2 = line;
    }

```



## handleboolean( ):

The input line is sent here if it contains the boolean operators (&&, !, or) – **we replaced the java || expression with (or) to avoid misleading the lexer** – and processed as follows.

- 1) If the input line contains (&&), we break the statement to small if condition statements.  
Ex: if( x>0 && x<6 ) → if(x>0) , if(x<6)  
then we pass it to the ifCondition method.
- 2) If the input line contains (or) we do the exact same steps done above but with a small adjustment, before sending the if statement we reverse the relop expression. Ex: if(x>0) → if(x<0)  
If we have multiple statements, we reverse all of them except for the last one.
- 3) Not is also the same.

```
private void handleBoolean(String program) throws IOException {
    program = program.replaceAll("if", "");
    if (program.contains("&&")) // expression has "&&"
    {
        String[] temp = program.split("&&");
        for (int i = 0; i < temp.length; i++) {
            temp[i] = temp[i].replaceAll("\\(", "");
            temp[i] = temp[i].replaceAll("\\)", "");
            temp[i] = temp[i].trim();
            temp[i] = "if ( " + temp[i] + " ) ";
            isBoolean = true;
            ifCondition(temp[i]);
        }
    }
}
```

Figure 11 and

```
String[] temp = program.split("or");
for (int i = 0; i < temp.length; i++) {
    temp[i] = temp[i].replaceAll("\\(", "");
    temp[i] = temp[i].replaceAll("\\)", "");
    temp[i] = temp[i].trim();
    if (i < temp.length - 1) {
        if (temp[i].contains(">"))
            temp[i] = temp[i].replaceAll(">", "<");
        else if (temp[i].contains("<"))
            temp[i] = temp[i].replaceAll("<", ">");
        else if (temp[i].contains("<="))
            temp[i] = temp[i].replaceAll("<=", ">=");
        else if (temp[i].contains(">="))
            temp[i] = temp[i].replaceAll(">=", "<=");
        else if (temp[i].contains("!="))
            temp[i] = temp[i].replaceAll("!= ", "==" );
        else if (temp[i].contains("=="))
            temp[i] = temp[i].replaceAll("== ", "!=" );
    } else {
        isLast = true;
    }
    temp[i] = "if ( " + temp[i] + " ) ";
    isBoolean = true;
    ifCondition(temp[i]);
}
```

Figure 12 or

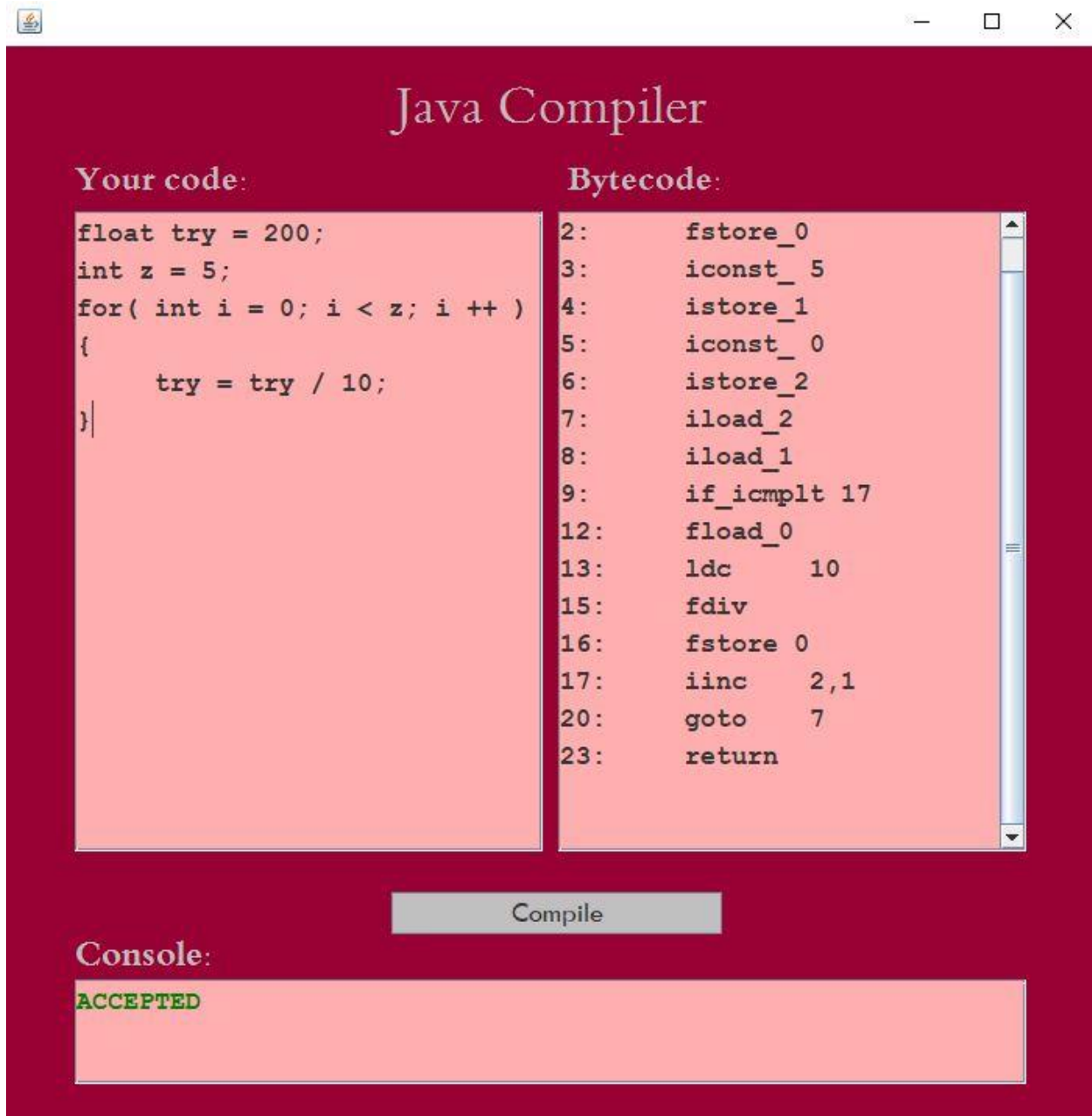
**Our compiler supports limited Boolean expressions Ex:**

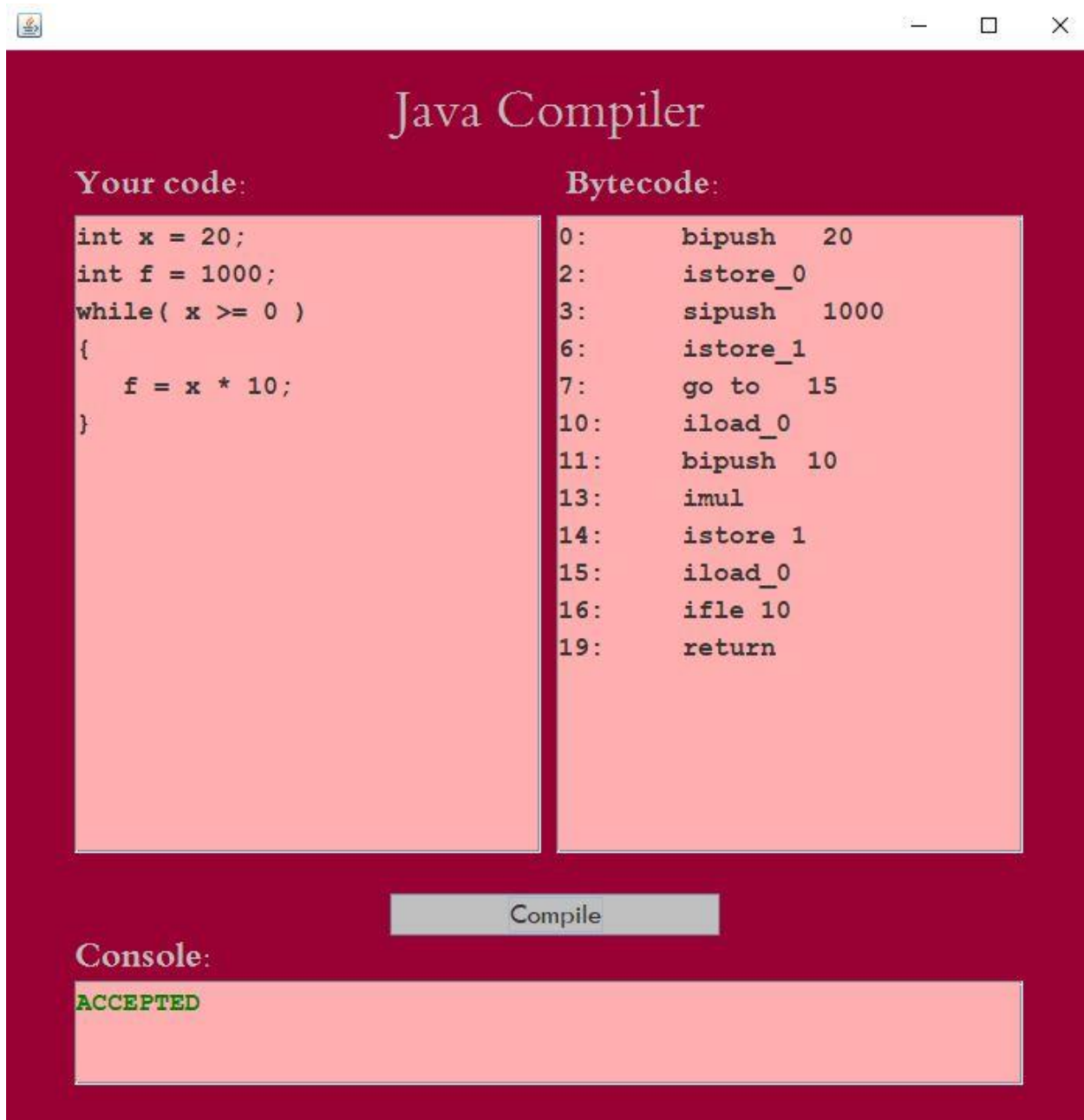
- It doesn't support mixed Boolean expressions yet. Ex: if( x<0 && x==4 **or** x>9 .... ).



## Sample Runs:

We added a simple GUI to make it user friendly.





The image shows a Java Compiler application window. It has a title bar with standard window controls. The main area is divided into three sections: 'Your code:', 'Bytecode:', and 'Console:'. The 'Your code:' section contains a Java program that initializes variables x and f, and enters a while loop. The 'Bytecode:' section shows the corresponding JVM instructions. The 'Console:' section shows the output 'ACCEPTED'. A 'Compile' button is located between the 'Bytecode:' and 'Console:' sections.

## Java Compiler

**Your code:**

```
int x = 20;
int f = 1000;
while( x >= 0 )
{
    f = x * 10;
}
```

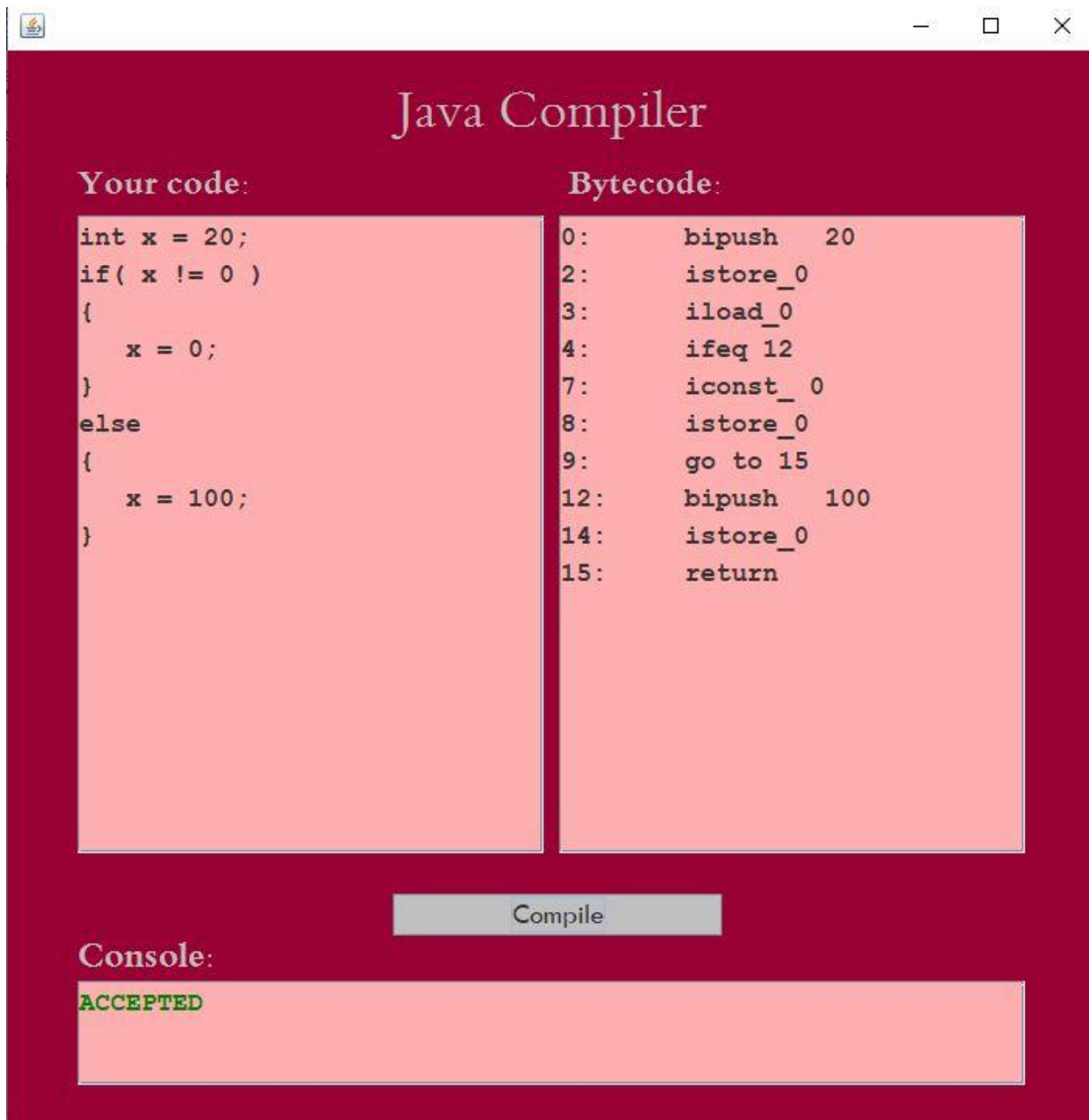
**Bytecode:**

```
0:    bipush    20
2:    istore_0
3:    sipush    1000
6:    istore_1
7:    go to     15
10:   iload_0
11:   bipush    10
13:   imul
14:   istore 1
15:   iload_0
16:   ifle 10
19:   return
```

**Console:**

ACCEPTED

Compile



The image shows a Java Compiler application window. It has a title bar with standard window controls (minimize, maximize, close) and a small icon on the left. The main area is divided into three sections: 'Your code:', 'Bytecode:', and 'Console:'. The 'Your code:' section contains a Java snippet. The 'Bytecode:' section shows the corresponding bytecode instructions. The 'Console:' section shows the output 'ACCEPTED'. A 'Compile' button is located between the 'Bytecode:' and 'Console:' sections.

## Java Compiler

**Your code:**

```
int x = 20;  
if( x != 0 )  
{  
    x = 0;  
}  
else  
{  
    x = 100;  
}
```

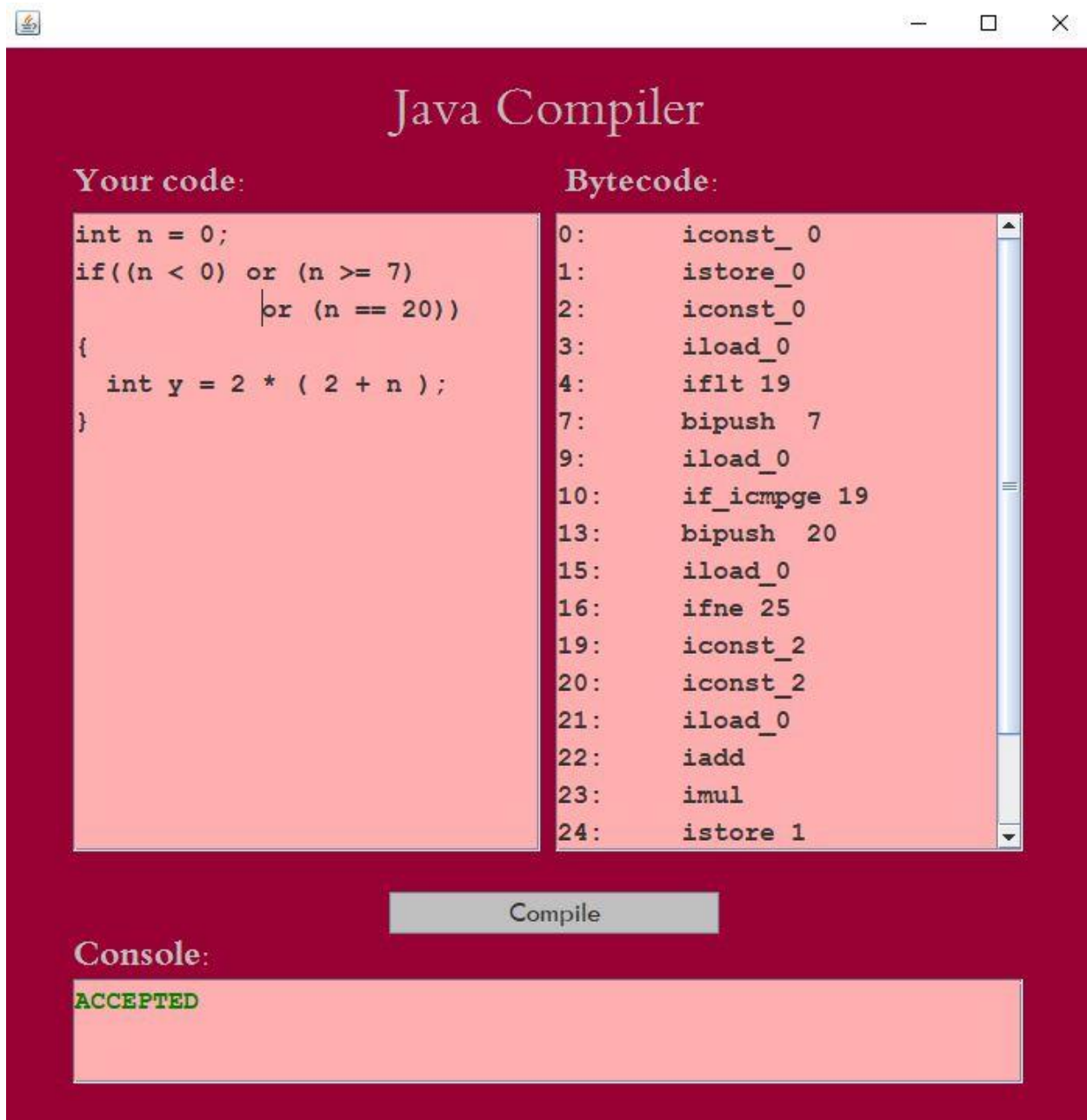
**Bytecode:**

```
0:    bipush    20  
2:    istore_0  
3:    iload_0  
4:    ifeq 12  
7:    iconst_0  
8:    istore_0  
9:    go to 15  
12:   bipush    100  
14:   istore_0  
15:   return
```

**Compile**

**Console:**

```
ACCEPTED
```



The image shows a Java Compiler window with a dark red background. It is divided into three main sections: 'Your code:', 'Bytecode:', and 'Console:'. The 'Your code:' section contains a Java snippet. The 'Bytecode:' section shows the corresponding JVM instructions. The 'Console:' section shows the output 'ACCEPTED'. A 'Compile' button is located between the 'Bytecode:' and 'Console:' sections.

## Java Compiler

**Your code:**

```
int n = 0;
if((n < 0) or (n >= 7)
    or (n == 20))
{
    int y = 2 * ( 2 + n );
}
```

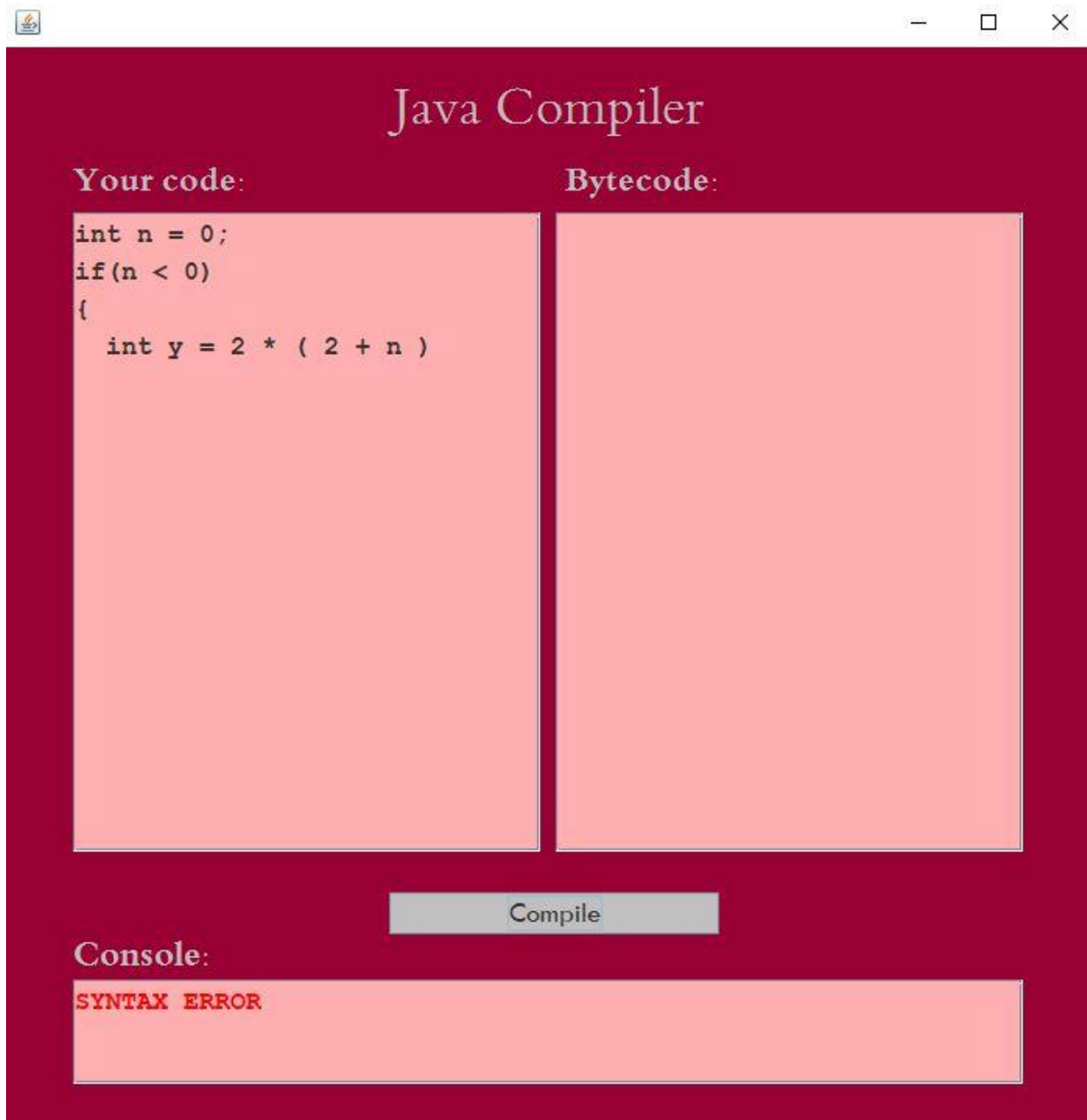
**Bytecode:**

```
0:   iconst_0
1:   istore_0
2:   iconst_0
3:   iload_0
4:   iflt 19
7:   bipush 7
9:   iload_0
10:  if_icmpge 19
13:  bipush 20
15:  iload_0
16:  ifne 25
19:  iconst_2
20:  iconst_2
21:  iload_0
22:  iadd
23:  imul
24:  istore 1
```

**Compile**

**Console:**

```
ACCEPTED
```



- Missing semi-column
- Missing bracket - } -