Programming Languages Translation

Phase 1 : Lexical Analyzer Generator

Name	ID
Mohamed Harraz	4608
Sherif Rafik	4635
Omar Nasr	4730

Project Objective:

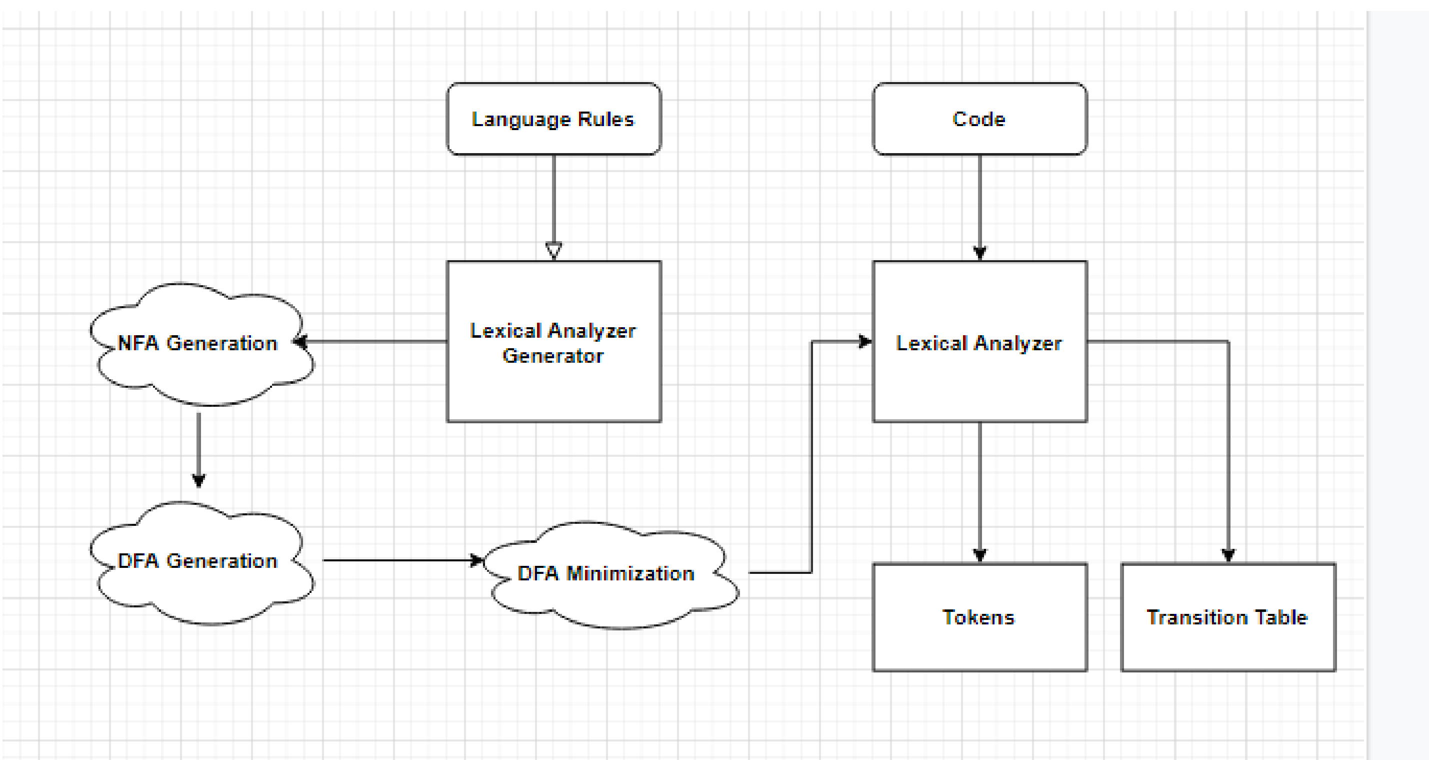
The aim of this phase is to practice techniques for building automatic lexical analyzer generation tools.

Project description:

The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens. The tool is required to construct a nondeterministic finite automata (NFA) for the given regular expressions, combine these NFAs together with a new starting state, convert the resulting NFA to a DFA, minimize it and emit the transition table for the reduced DFA together with a lexical analyzer program that simulates the resulting DFA machine.

The generated lexical analyzer has to read its input one character at a time, until it finds the longest prefix of the input, which matches one of the given regular expressions. It should create a symbol table and insert each identifier in the table. If more than one regular expression matches some longest prefix of the input, the lexical analyzer should break the tie in favor of the regular expression listed first in the regular specifications

Project flow:



Data Structures:

HashMaps:

A Java built-in data structure that stores data in (key, value) pairs. To access a value, one must know its key.

ArrayLists:

A Java built-in data structure that presents a dynamic array.

Stack:

A Java built-in data structure that's based on the LIFO (last-in-first-out) principle.

Node:

A data structure representing a graph node, carries a **HashMap** whose keys are transition characters and values are an **ArrayList** of nodes.

A node can be a state node or an end node (acceptance node).

Each node has a unique id.

Graph:

A data structure, on creation consists of two nodes, a starting node, and an ending node (acceptance node).

Graph represents the non-deterministic finite automata, and the deterministic finite automata

Pair:

A Java built-in data structure that stores data in (key, value) manner.

Algorithms and techniques:

Infix to postfix:

Infix expression: The expression is of the form a operation b, where the operator is in-between every pair of operands.

Postfix expression: The expression is of the form a b operation, where the operation is comes after the pair of operands.

I use the infix to postfix algorithm to convert the regular expressions into postfix expressions, because the regular expression is scanned from left to right.

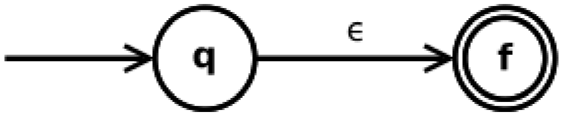
Code snippet:

```
public static ArrayList<String> infixToPostFix(ArrayList<String> expression) {

   ArrayList<String> result = new ArrayList<String>();

   Stack<String> stack = new Stack<String>();

   for (int i = 0; i < expression.size(); i++) {
      String c = expression.get(i);

      if (precedence(c) > 0) {
         while (!stack.isEmpty() && precedence(stack.peek()) >= precedence(c))
            result.add(stack.pop());
         stack.push(c);
      } else if (c.equals(")")) {
         if (expression.size() != 1) {
            while (!stack.isEmpty() && !(stack.peek().equals("("))) {
               result.add(stack.pop());
            }
            stack.pop();
         } else {
            stack.push(c);
         }
      } else if (c.equals("(")) {
         stack.push(c);
      } else { // Character is neither operator nor (
         result.add(c);
      }

   }

   while (!stack.isEmpty())
      result.add(stack.pop());

   return result;
}
```

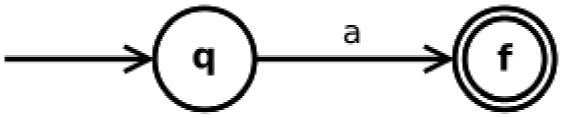
Regular Expression to NFA – Thompson's construction:

To convert the regular expressions to NFA, Thompson's construction algorithm is used.

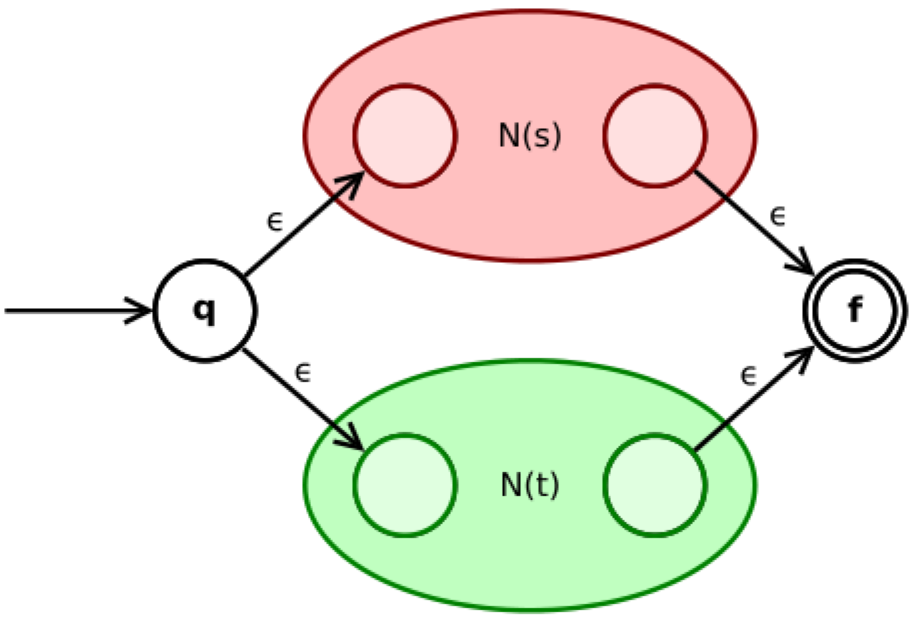
The **empty expression** (epsilon) is converted to



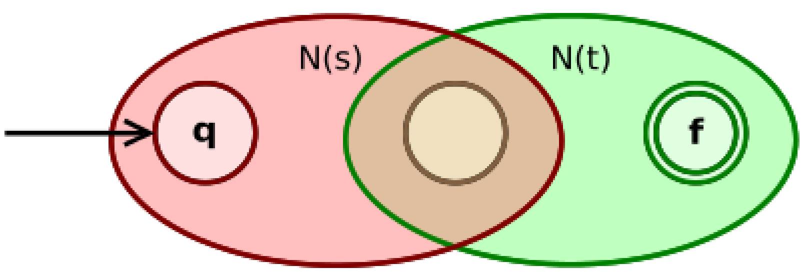
A symbol **a** of the input alphabet is converted to



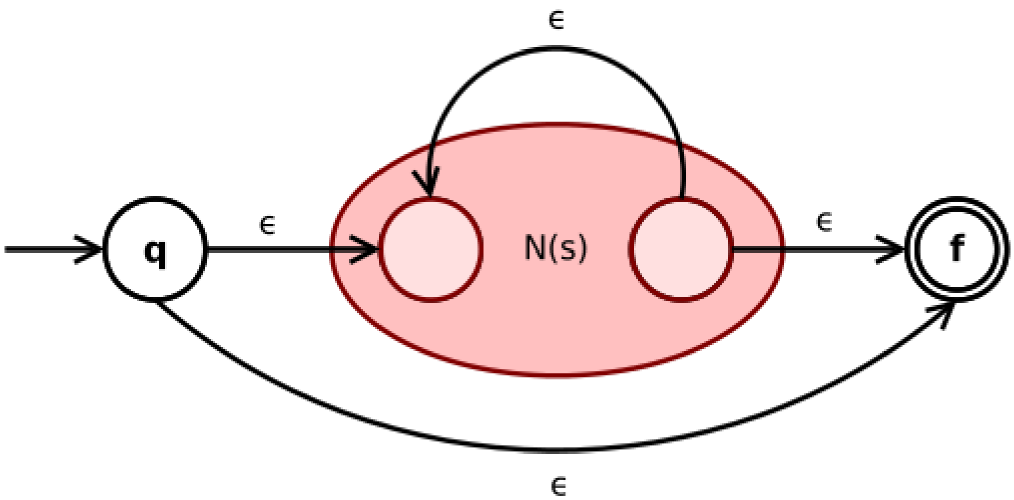
The union expression $s|t$ is converted to



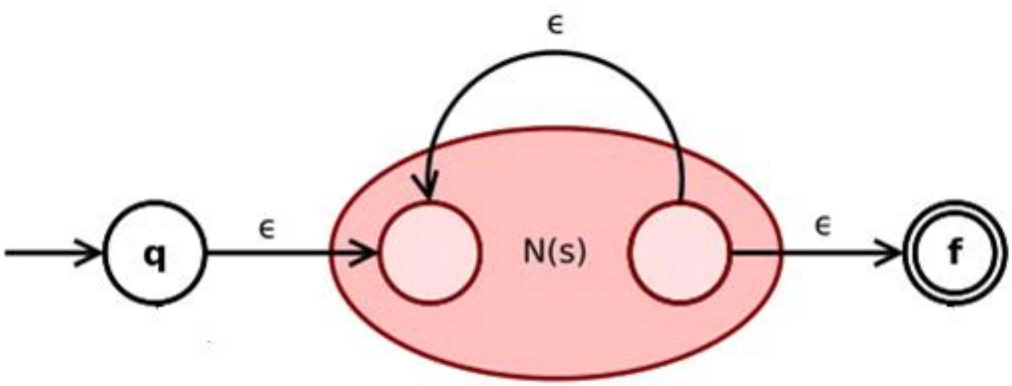
The concatenation expression st is converted to



The Kleene closure expression is converted to



The star closure expression is converted to



Code snippet:

```
private Graph createNfa(ArrayList<String> expression) {
    // create a stack
    Stack<Graph> nfa = new Stack<Graph>();
    // Scan all characters one by one
    for (int i = 0; i < expression.size(); i++) {
        String currentExpression = expression.get(i);
        if (NfaUtility.isRegexOperator(currentExpression)) {
            if (currentExpression.equals(Constant.KLEENE)) {
                Graph g = nfa.pop();
                nfa.push(GraphUtility.kleeneClosure(g));
            } else if (currentExpression.equals(Constant.PLUS)) {
                Graph g = nfa.pop();
                nfa.push(GraphUtility.plusClosure(g));
            } else if (currentExpression.equals(Constant.OR)) {
                Graph right = nfa.pop();
                Graph left = nfa.pop();
                nfa.push(GraphUtility.or(right, left));
            } else if (currentExpression.equals(Constant.CONCATENATE)) {
                Graph right = nfa.pop();
                Graph left = nfa.pop();
                nfa.push(GraphUtility.concatenate(left, right));
            }
        } else {
            if (definitionNfa.containsKey(currentExpression)) {
                Graph g = new
Graph(definitionNfa.get(currentExpression));
                nfa.push(g);
            } else if (symbols.contains(currentExpression) &&
!currentExpression.equals("\\L")) {
                String nodeName = expression.get(i).substring(1);
                nfa.push(new Graph(nodeName));
            } else {
                nfa.push(new Graph(currentExpression));
            }
        }
    }
    return nfa.pop();
}
```

NFA to DFA – Subset construction:

To convert NFA to DFA,

Code snippet:

```
private void constructDFA(Graph NFACombined) {
   while (!DFAStatesUnmarked.empty()) { /** while there is unmarked states */
      ArrayList<Node> T = DFAStatesUnmarked.pop(); /** mark */
      String TsID = DfaUtility.createUnionID(T);// ID,ID,ID
      ArrayList<Node> U;
      for (String a : DfaUtility.getUnionInputs(T)) { /** for all possible inputs a */
         U = epsilonClosure(move(T, a)); /** U (a new DFA state) = */
         String newNodeTypes = DfaUtility.getNodeType(U);
         String newID = DfaUtility.createUnionID(U);
         if (!DFATransTable.containsKey(newID)) { /** if U is new add to Unmarked and transition
table */
            DFAStatesUnmarked.push(U);
            Node node = new Node();
            node.setNodeTypes(newNodeTypes);
            DFATransTable.put(newID, node);
            if (U.contains(NFACombined.getDestination()))
               node.setEnd(true);
         }
         DFATransTable.get(TsID).addEdge(a, DFATransTable.get(newID)); /** transition table [T , a]
= U */
      }
   }
}
private ArrayList<Node> epsilonClosure(ArrayList<Node> T) {
   Stack<Node> stack = new Stack<>();
   ArrayList<Node> epsilonClosureOut = new ArrayList<>();
   epsilonClosureOut = T;
   /** push all states of T onto stack */
   for (Node node : epsilonClosureOut) {
      stack.push(node);
   }
   /** while stack is not empty pop first element t */
   while (!stack.empty()) {
      Node t = stack.pop();
      HashMap<String, ArrayList<Node>> neighbours = new HashMap<>();
      neighbours = t.getMap();
      /**
       * search for all unvisited (not in epsilonClosure) nodes reachable from t
       * through an epsilon edge
       */
      for (String key : neighbours.keySet()) {
         if (key.equals(Constant.EPSILON)) {
            for (Node node : neighbours.get(key)) {
               if (!epsilonClosureOut.contains(
                     node)) { /** if not already in epsilonClosureOut add it and push it to the
stack */
                  epsilonClosureOut.add(node);
                  stack.push(node);
               }
            }
         }
      }
   }
   return epsilonClosureOut;
}
```

Subset construction algorithm is used.

DFA minimization:

Code snippet:

```
private void minimizeDFA(DFA DFA) {
   HashMap<Integer, Integer> nodeParents = new HashMap<>(); /** int node -> int 
parent */
   HashMap<String, Node> DFATransTable = DFA.getDFATransTable();
   HashMap<Integer, ArrayList<Node>> grouping = new HashMap<>();
   ArrayList<Node> nonAcceptingState = new ArrayList<>();
   ArrayList<Node> acceptingState = new ArrayList<>();
   for (String string : DFATransTable.keySet()) {
      Node node = DFATransTable.get(string);
      if (node.isEnd())
         acceptingState.add(node);
      else
         nonAcceptingState.add(node);
   }

   grouping.put(nonAcceptingState.get(0).getCurrentId(), nonAcceptingState);
   while (!acceptingState.isEmpty()) {
      ArrayList<Node> partition = new ArrayList<>();
      for (int i = 1; i < acceptingState.size(); i++) {
         if 
(acceptingState.get(i).getNodeTypes().equals(acceptingState.get(0).getNodeTypes(
))) {
            partition.add(acceptingState.remove(i));
            i--;
         }
      }
      partition.add(acceptingState.remove(0));
      initNodeParents(partition, nodeParents);
      grouping.put(partition.get(0).getCurrentId(), partition);
   }

   initNodeParents(nonAcceptingState, nodeParents);
   HashMap<Integer, ArrayList<Node>> newGrouping = grouping;
   do {
      grouping = newGrouping; /** last grouping */
      newGrouping = constructGroupings(grouping, nodeParents); /** new grouping 
*/
   } while (newGrouping.size() != grouping
         .size()); /**
                    * while last groupings not the same as the new groupings 
continue to minimize
                    */
   linkDFAFinalGroupings(newGrouping, DFA);
}
```

Tokenization:

Code snippet:

```
public ArrayList<Pair<String, String>> getTokens(String input) {
   savedLexems = new ArrayList<>();
   Node start = minimalDFA.getInitialNode();
   int idx = 0;
   int retValue;
   do {
      retValue = addGenerations(input, idx, idx, savedLexems, start);
      if (retValue == -1)
         return null;

      if (retValue == -2)
         idx++;
      else
         idx = retValue + 1;
   } while (idx < input.length());
   return savedLexems;
}

private int addGenerations(String input, int startIdx, int idx,
ArrayList<Pair<String, String>> lexems,
      Node currNode) {
   if (idx >= input.length())
      return -2;
   char currentChar = input.charAt(idx);
   if (currentChar == ' ' || currentChar == '\n' || currentChar == '\r' ||
currentChar == '\t')
      return -2;
   String transition = Integer.toString(currNode.getCurrentId()) +
Constant.SEPARATOR + input.charAt(idx);
   Pair<Node, String> nextTransition = transitionTable.get(transition);
   if (nextTransition != null) {
      String acceptanceStates[] =
nextTransition.getValue().split(Constant.SEPARATOR);
      String acceptance = getAcceptanceState(acceptanceStates,
input.substring(startIdx, idx + 1));
      int retValue = addGenerations(input, startIdx, idx + 1, lexems,
nextTransition.getKey());
      if (retValue == -1 || retValue == -2) {
         if (acceptance.equals(""))
            return -1;
         lexems.add(new Pair<>(input.substring(startIdx, idx + 1), acceptance));
         return idx;
      } else {
         return retValue;
      }
   }
   return -1;
}
```

Assumptions:

When adding the concatenating symbol in the regular expression we use back tick **"`"**

When grouping states together in the DFA, the parent state is the name of the children states separated by commas ","

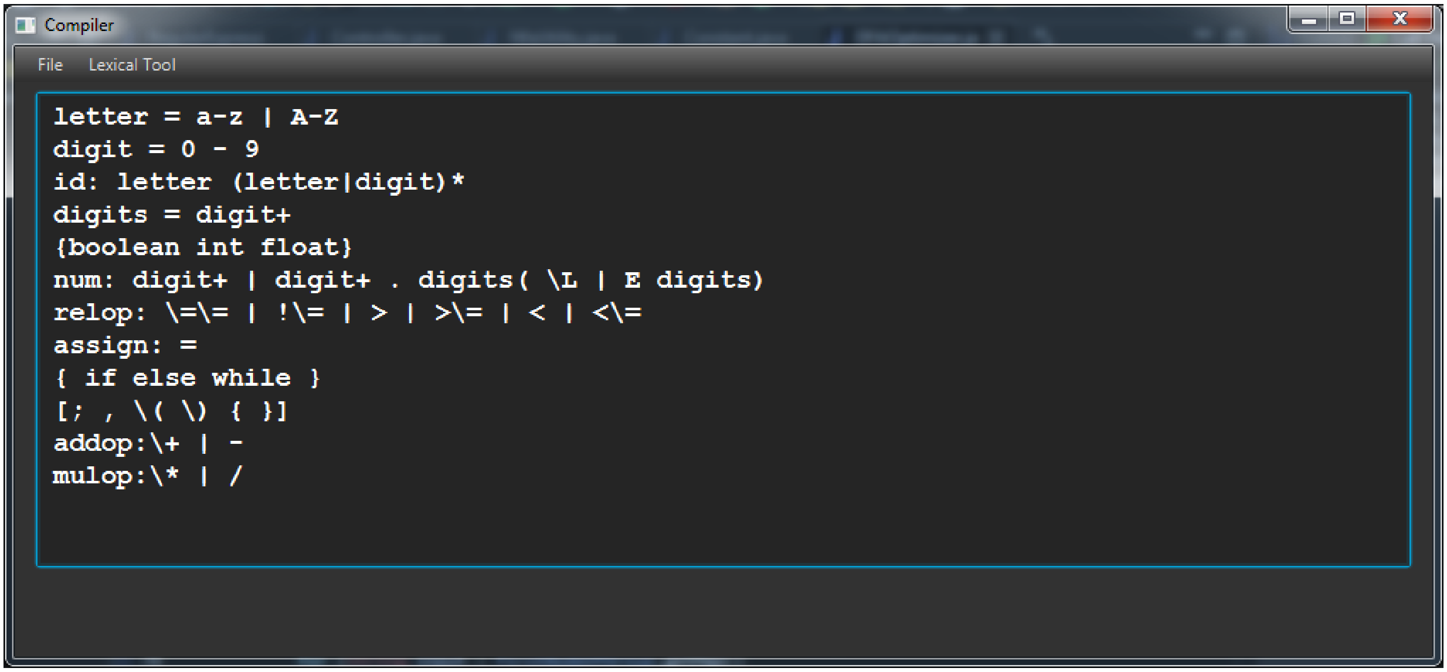
Project structure:

Project follows the MVC (model-view-controller) design pattern.

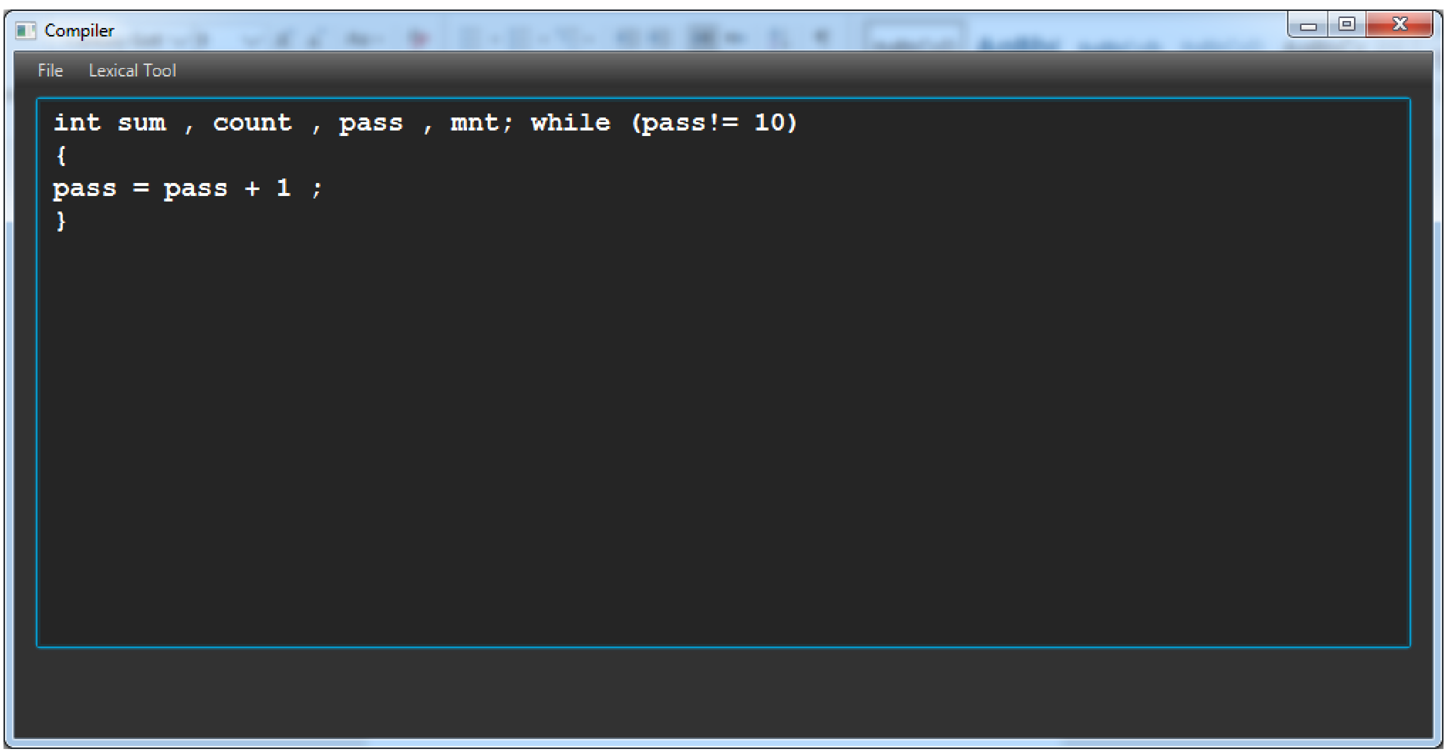
1- Every GUI related class is found in the view package
2- Every Logic related class is found in the model package
   a. Construction
      i. Line Processor class
      ii. Rules Container class
   b. DFA
      i. DFA class
      ii. DFA Optimizer class
   c. Graph
      i. Graph class
      ii. Node class
   d. NFA
      i. Keyword class
      ii. NFA class
      iii. Punctuation class
      iv. Regular Definition class
      v. Regular Expression class
3- Controller class is found in the controller package
4- Helper classes are found in the utilities package

Sample runs:

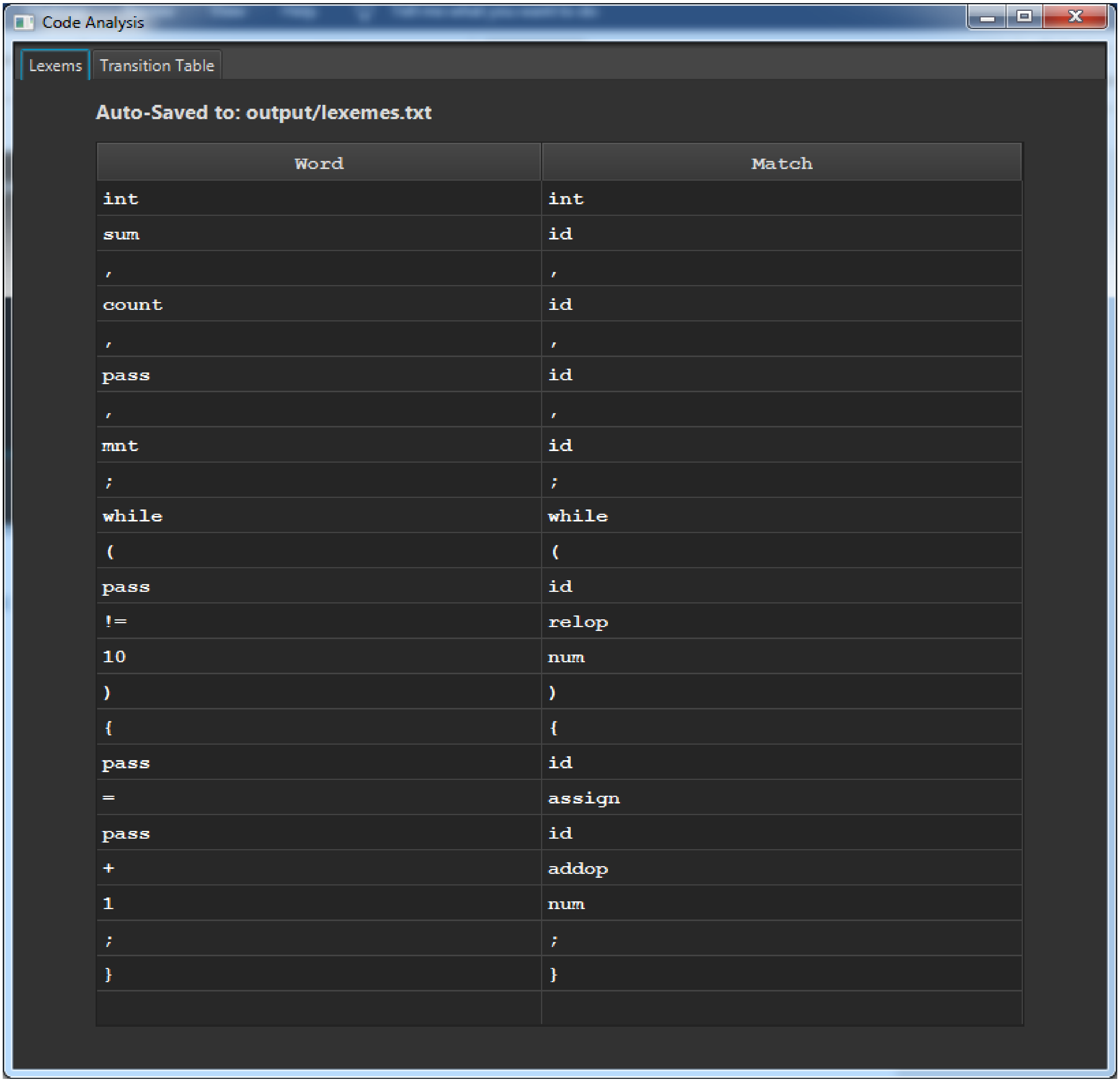
Language rules:



Program:



Tokens classes:



Word	Match
int	int
sum	id
,	,
count	id
,	,
pass	id
,	,
mnt	id
;	;
while	while
(	(
pass	id
!=	relop
10	num
)	)
{	{
pass	id
=	assign
pass	id
+	addop
1	num
;	;
}	}

Part of the transition table:

Code Analysis

Lexems | Transition Table

Source Node ID	Input	Destination Node ID	Possible Output
1546	z	1556	id
1568	=	1573	relop
1546	0	1556	id
1546	1	1556	id
1546	2	1556	id
1546	3	1556	id
1546	4	1556	id
1546	5	1556	id
1546	6	1556	id
1546	7	1556	id
1546	8	1556	id
1546	9	1556	id
1546	Z	1556	id
1546	a	1556	id
1546	b	1556	id
1546	c	1556	id
1546	d	1556	id
1546	e	1556	id
1546	f	1556	id
1546	g	1556	id
1546	h	1556	id
1546	i	1556	id
1546	j	1556	id
1546	k	1556	id
1546	l	1547	id
1546	m	1556	id
1546	n	1556	id
1546	o	1556	id
1546	p	1556	id
1546	q	1556	id

First test case:

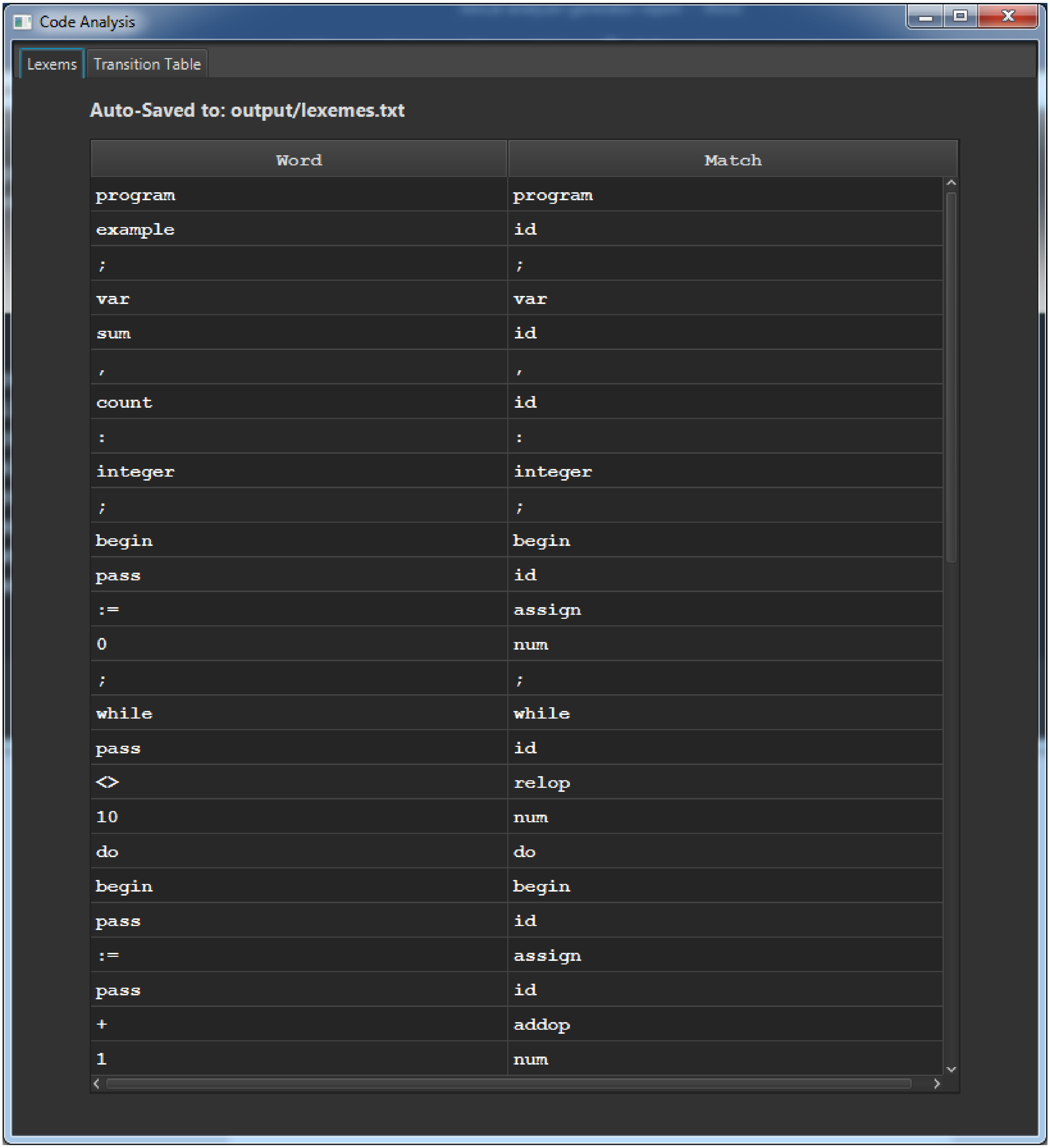
Language rules:

```
letter = a-z | A-Z
digit = 0 - 9
digits = digit+
{program var integer}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \= | <> | > | >\= | < | <\=
assign: \:\=
{real begin end if else then while do read write}
addop: \+ | -
mulop: \* | /
[: ; , . \( \)]
id: letter (letter|digit)*
```

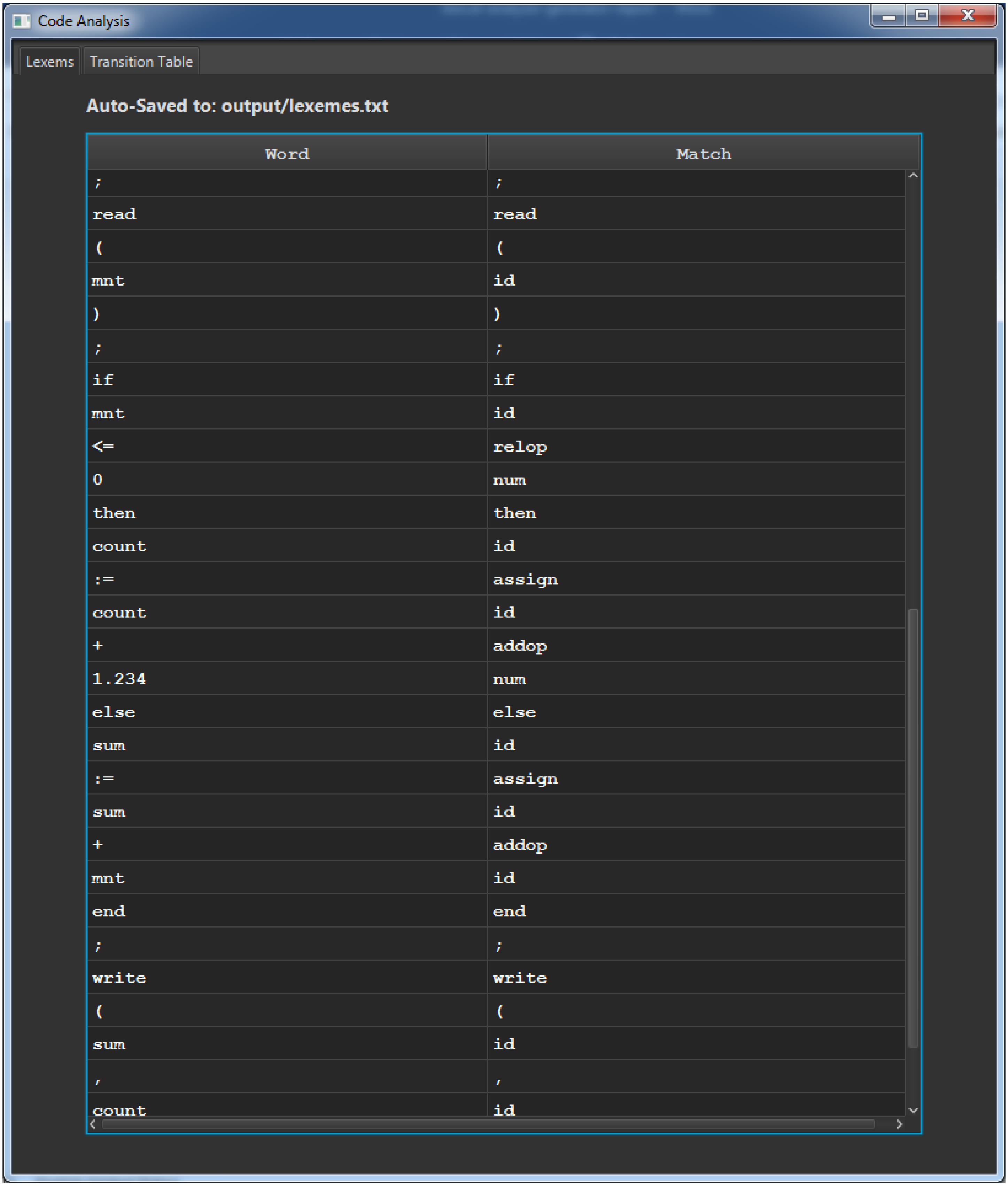
Program:

```
program example ;
var  sum , count  : integer ;
        begin
                pass := 0 ;
                while pass <> 10  do
                begin
                pass := pass + 1 ;
read (mnt) ;
if mnt <= 0 then
count := count + 1.234
else
sum := sum + mnt
end ;
write (sum , count)
end.
```

Tokens classes:



Word	Match
program	program
example	id
;	;
var	var
sum	id
,	,
count	id
:	:
integer	integer
;	;
begin	begin
pass	id
:=	assign
0	num
;	;
while	while
pass	id
<>	relop
10	num
do	do
begin	begin
pass	id
:=	assign
pass	id
+	addop
1	num


Code Analysis
Lexems
Transition Table
Auto-Saved to: output/lexemes.txt
Word
Match
; ;
read read
((
mnt id
))
; ;
if if
mnt id
<= relop
0 num
then then
count id
:= assign
count id
+ addop
1.234 num
else else
sum id
:= assign
sum id
+ addop
mnt id
end end
; ;
write write
((
sum id
, ,
count id

Part of the transition table:

Code Analysis

Lexems | Transition Table

Source Node ID	Input	Destination Node ID	Possible Output
1799	5	1771	id
1799	4	1771	id
1799	3	1771	id
1799	2	1771	id
1799	1	1771	id
1799	0	1771	id
1827	6	1827	num
1827	7	1827	num
1827	4	1827	num
1827	5	1827	num
1827	8	1827	num
1827	9	1827	num
1827	.	1835	
1827	2	1827	num
1827	3	1827	num
1827	0	1827	num
1827	1	1827	num
1800	B	1771	id
1800	A	1771	id
1800	D	1771	id
1800	C	1771	id
1800	F	1771	id
1800	E	1771	id
1800	H	1771	id
1800	G	1771	id
1800	9	1771	id
1800	R	1771	id
1800	Q	1771	id
1800	T	1771	id