Programming Languages Translation

Front-end compiler

Name	ID
Mohamed Harraz	4608
Sherif Rafik	4635
Omar Nasr	4730

Prof. Dr/ Nagia M. Ghanem

Table of Contents

Abstract: **4**

Objectives: **4**

Introduction: **5**

1- Lexical Analyzer 6

2- Syntax Analyzer: 6

3- Semantic Analyzer: 6

4- Intermediate code generator: 6

Problem description: **7**

Project Flow: **8**

Data structures: **9**

1- Lexical Analyzer 9

2- Parser generator 9

3- Java bytecode generator 9

4- Common 10

Explanation of all algorithms and techniques: **10**

1- Shunting-yard algorithm 10

2- Regular Expression to NFA – Thompson's construction: 11

3- NFA to DFA – Subset construction: 13

4- DFA minimization: 14

5- Left Recursion Elimination: 15

6- Left Factoring: 16

First Construction: 16

7- Follow Construction: 17

8- Parsing table construction: 18

9- Parse: 19

10- Backpatching: 20

Sample runs: **22**

1- Lexical Analyzer 22

2- Parser generator 28

3- Code generation 34

Comments about tools: **39**

1- Graph editor: 39

2- Flex (fast lexical analyzer generator): 39
3- Jasmin: 40
4- Bison: 41
Project internal structure: 42
Assumptions: 43
Conclusion: 43
Role of each student: 44
References: 45

Abstract:

A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language without changing the meaning of the program through a number of stages.

This paper exposes a front-end compiler implementation using JAVA and C++ (Bison and Flex) programming languages. At the beginning the paper gives a brief introduction to the problem description and the objective of this implementation, then the paper provides a description for the project flow and explains the various data structures and algorithms used to implement each phase.

Finally, the paper concludes with some sample runs and our conclusion for implementing the front-end compiler.

Objectives:

Our objective is to practice various algorithms and tools that help in building:

1- A lexical analyzer generator
2- A parser generator
3- A code generator for the JAVA programming language

Introduction:

A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language without changing the meaning of the program through a number of stages.

The compiler has two modules namely front end and back end. Front-end constitutes of the lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. And the rest are assembled to form the back end.

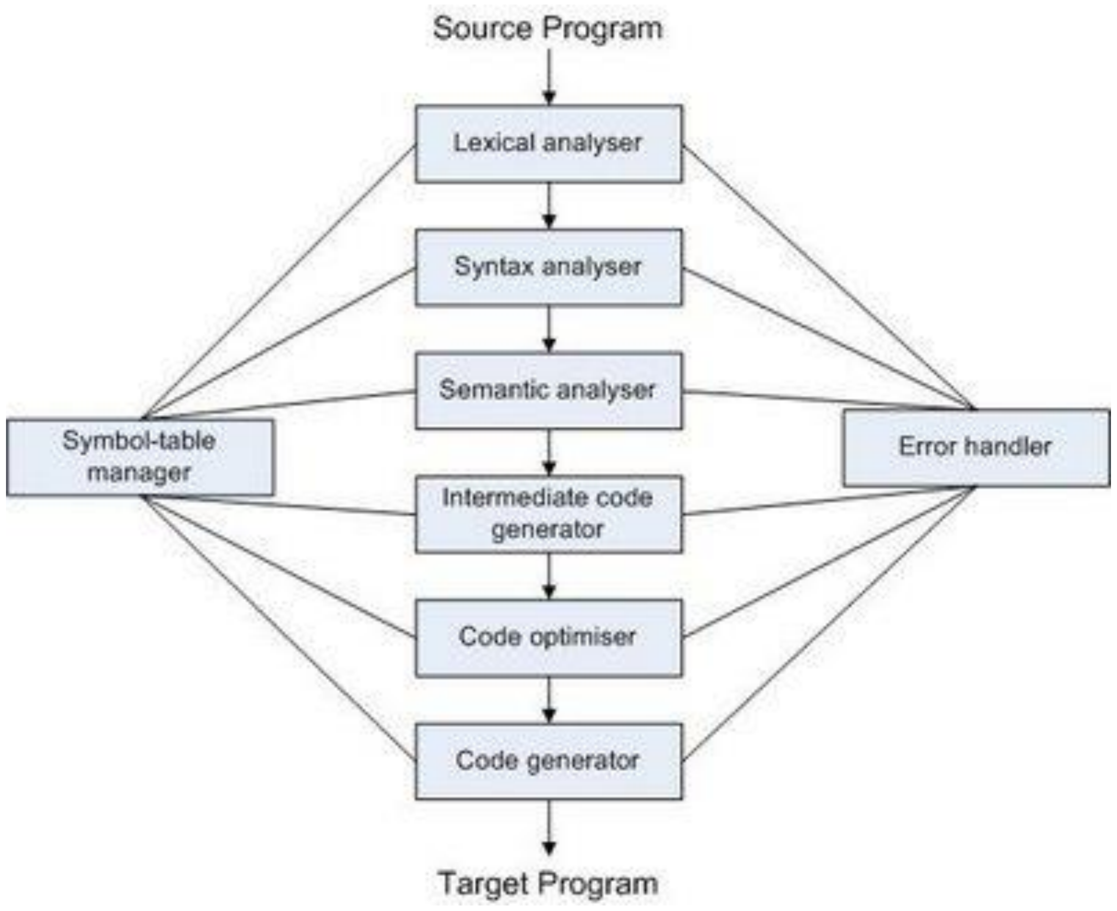


Figure 1: Model of a compiler

(Compilers Principles, Techniques, and Tools, 2006)

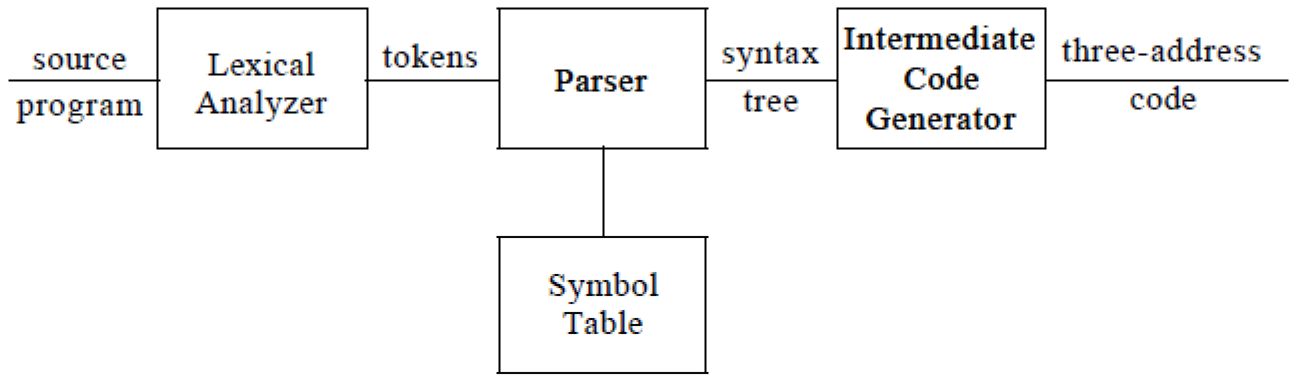


Figure 2: Model of a compiler front end

(Compilers Principles, Techniques, and Tools, 2006)

1- Lexical Analyzer:

Also known as the scanner, is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences and break these syntaxes into a series of lexemes. Each lexeme corresponds to a token.

2- Syntax Analyzer:

Also known as the parser, is the second phase of a compiler. It constructs the parse tree. It parses the lexemes one by one and uses context free grammar to generate the parse tree.

3- Semantic Analyzer:

It verifies whether the parse tree is correct or not. It also inserts the semantic rules to the nodes of the parse tree if found to be correct.

4- Intermediate code generator:

It generates intermediate code, that is a form of code that can be directly translated into its target machine code.

Example: Three address code

Intermediate code is converted to machine language using the last two phases of the compiler.

Problem description:

It is required to develop a compiler front-end that will convert java code to java bytecode, performing necessary lexical, syntax, semantic analysis.

We divide this project into 3 phases:

1- Lexical analyzer generator

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.

2- Parser generator

The parser generator expects an LL (1) grammar as input, it should compute the first and follow and uses them to construct the predictive parsing table.

The table is used to derive a predictive top-down parser. If the input grammar is not LL (1), an appropriate error message should be produced.

If an error is encountered, a panic-mode error recovery routine is to be called to print an error message and to resume parsing

3- Java bytecode generator

The java bytecode generator must follow bytecode instructions defined in the java virtual machine specifications. (Java SE Specification, n.d.)

Grammar is required to cover:

- Primitive types (int, float) with operations on them (+, -, *, /)
- Boolean Expressions
- Arithmetic Expressions
- Assignment statements
- If-else statements
- for loops
- while loops

Project Flow:

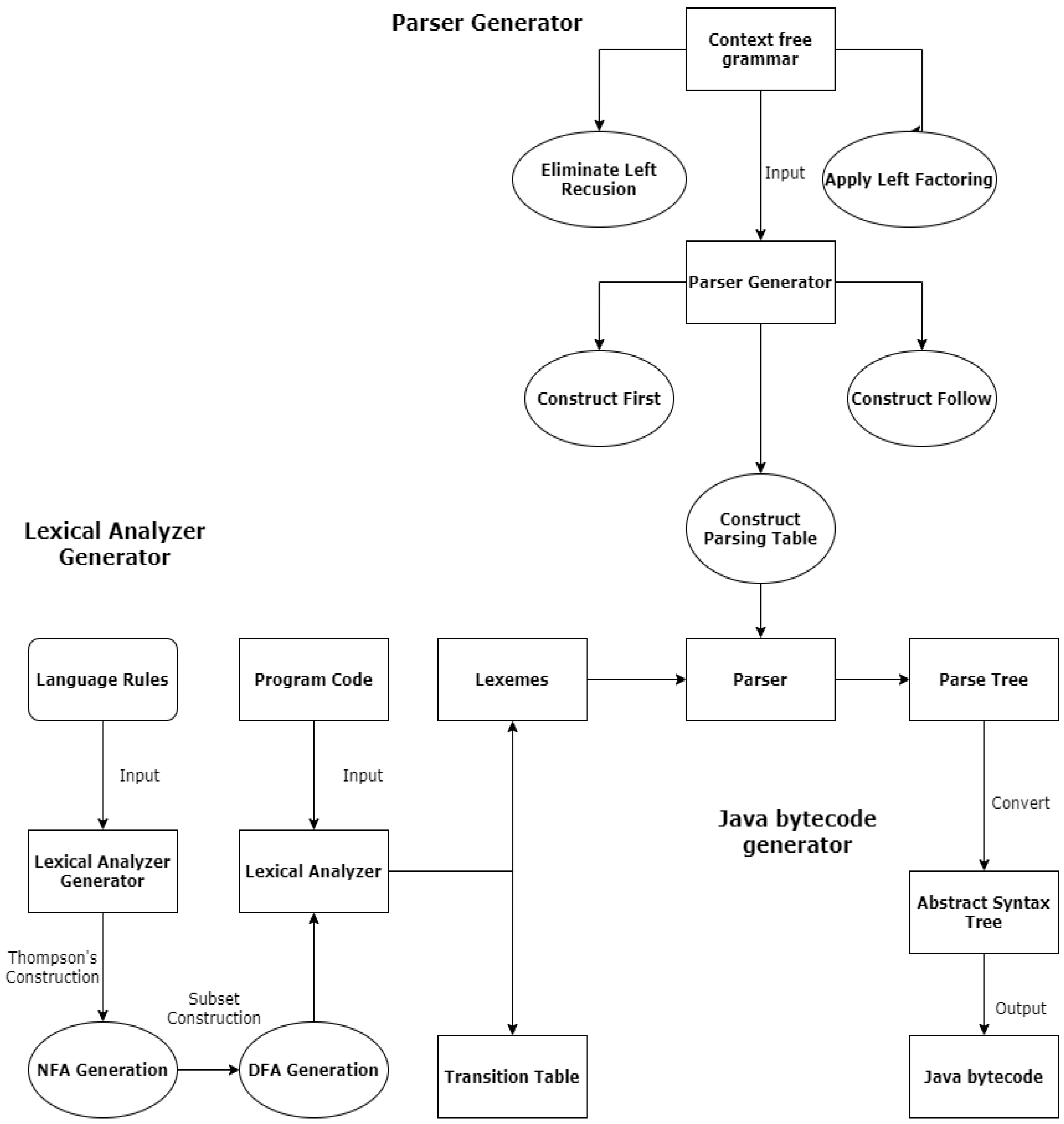


Figure 3: Project flow

Data structures:

1- Lexical Analyzer:

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.

2- Parser generator:

Trie:

A *data structure* which is used to store the collection of strings and makes searching of a pattern in words easier.

- Used in applying **left factoring** (if needed)

HashSets:

A Java built-in data structure that represents a set, where the input is hashed before insertion.

- Used in the construction of the **first** and **follow**

3- Java bytecode generator:

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

- Used in the symbol table implementation

Union:
In bison, union directive specifies the datatypes of the semantic rules for every possible type of terminals and non-terminals.
Vector:
A C++ built-in data structure that presents a dynamic array.

4- Common:

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.

Pair:

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

Explanation of all algorithms and techniques:

1- Shunting-yard algorithm:

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.

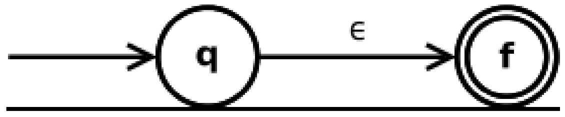
Pseudocode:

```
While there are tokens to be read:
  Read a token
  If it's a number add it to stack
  If it's an operator
    While there's an operator on the top of the stack with greater precedence:
        Pop operators from the stack onto the output queue
    Push the current operator onto the stack
 If it's a left bracket push it onto the stack
 If it's a right bracket
     While there's not a left bracket at the top of the stack:
         Pop operators from the stack onto the output queue.
     Pop the left bracket from the stack and discard it
While there are operators on the stack, pop them to the queue
```

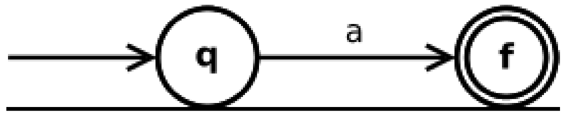
2- Regular Expression to NFA – Thompson's construction:

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

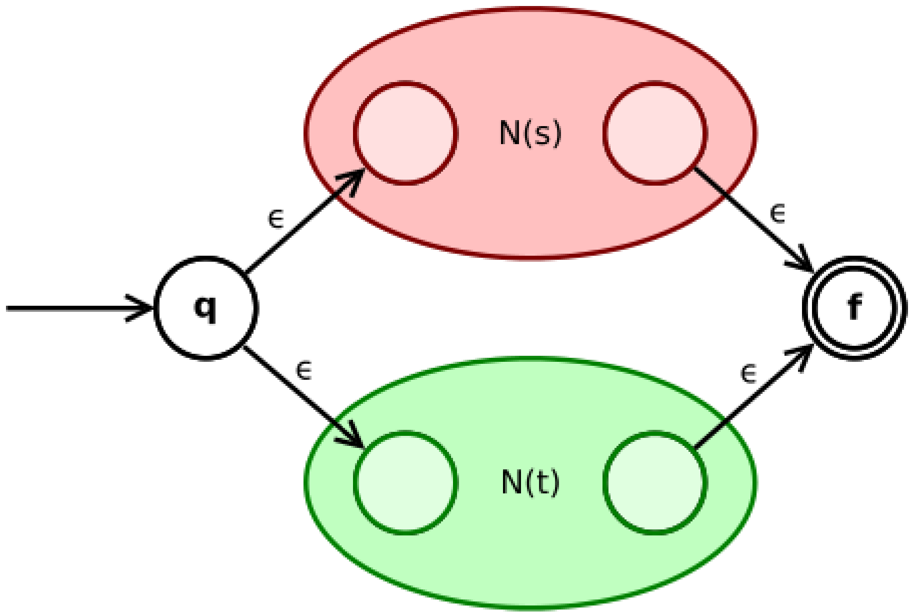
The **empty expression** (epsilon) is converted to:



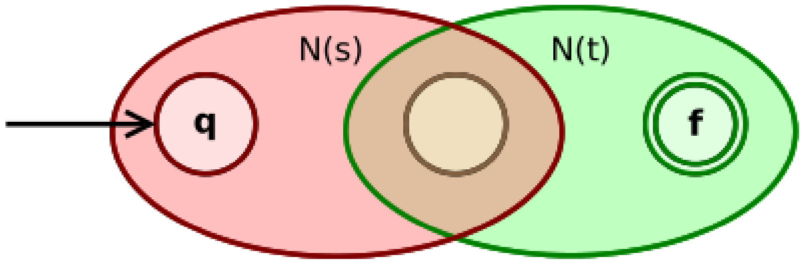
A symbol **a** of the input alphabet is converted to:



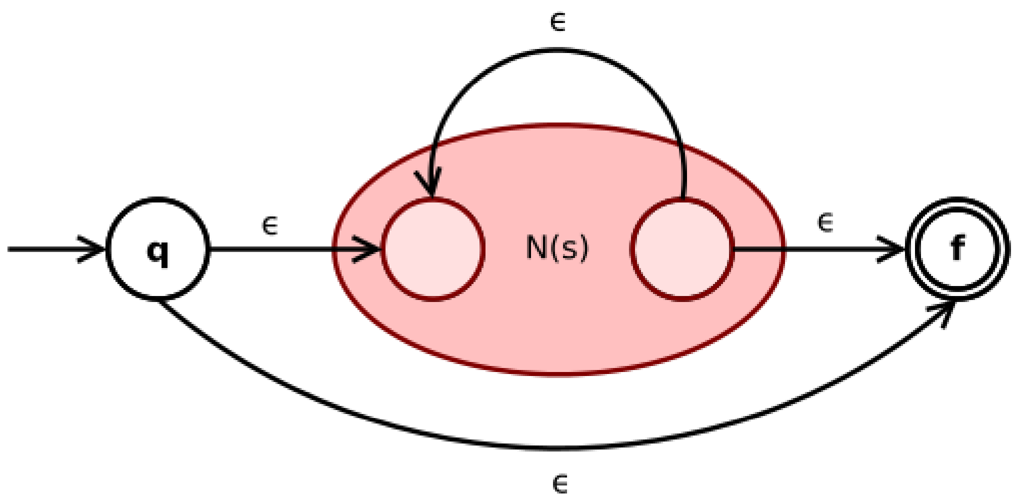
The **union** expression $s|t$ is converted to:



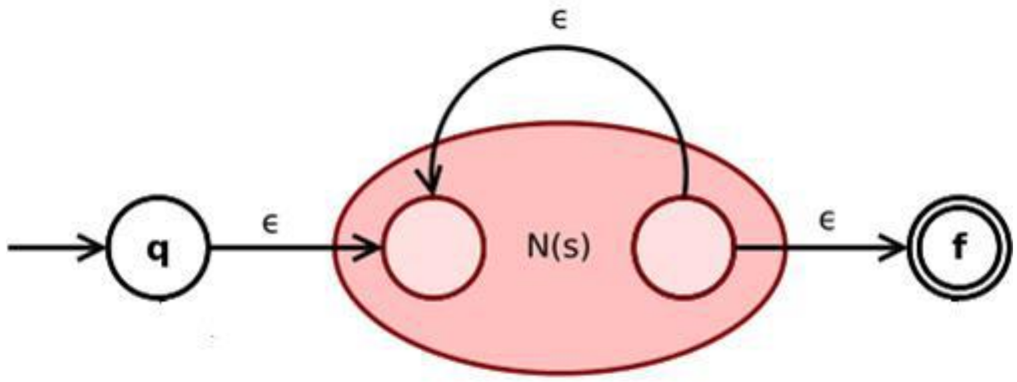
The **concatenation** expression st is converted to:



The **Kleene closure** expression is converted to:



The **star closure** expression is converted to:



(Compilers Principles, Techniques, and Tools, 2006) (Thompsons's Construction - Wikipedia)

3- NFA to DFA – Subset construction:

Using the subset construction algorithm, each NFA can be translated to an equivalent DFA that can recognize the same formal language.

In the transition table of NFA, each entry is a set of states. In DFA, each entry is a single state. The general idea NFA-to-DFA construction is that each state in the DFA corresponds to a set of states in the NFA.

For an NFA with epsilon moves, the construction must be modified to deal with these by computing the epsilon closures of the states.

Epsilon closure: The set of all state reachable from some given state using only epsilon moves.

We move from one state to another using the move algorithm.

Move (T, a): Set of states to which there is a transition on input **a** from some NFA state s in the set of states **T**

Pseudocode:

- Create the start state of the DFA by taking the epsilon-closure of the start state of the NFA.
- Perform the following for the new DFA state:
  For each possible input symbol:
    - Apply move to the newly-created state and the input symbol;
        - return a set of states.
    - Apply the epsilon-closure to this set of states, possibly resulting in a new set.
    - This set of NFA states will be a single state in the DFA.
- Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
- The finish states of the DFA are those which contain any of the finish states of the NFA.

(Compilers Principles, Techniques, and Tools, 2006) (Powerset construction - Wikipedia)

4- DFA minimization:

For each regular language, there also exists a **minimal automaton** that accepts it, that is, a DFA with a minimum number of states and this DFA is unique.
The minimal DFA ensures minimal computational cost for tasks such as pattern matching.
There are two classes of states that can be removed or merged from the original DFA without affecting the language, it accepts to minimize it:

- **Unreachable states** are the states that are not reachable from the initial state of the DFA, for any input string.
- **Non-Distinguishable states** are those that cannot be distinguished from one another for any input string.

Pseudocode:

- Initially start with two sets of states: the final, and the non-final states.
- For each state-set created by the previous iteration, examine the transitions for each state and each input symbol. If they go to a different state-set for any two states, then these should be put into different state-sets for the next iteration.
- Repeat the previous step until no new state-sets

5- Left Recursion Elimination:

A grammar is left recursive if it has a nonterminal A such that there is a derivation **A → Aα | β** where α and β are sequences of terminals and non-terminals that do not start with A.

A grammar can have direct left recursion having the form: **A → Aα | β** and indirect left recursion having the form: **A → Br, B → Cd, C → At.**

While designing a top down-parser, if the left recursion exists in the grammar then the parser falls in an infinite loop, here because A is trying to match A itself, which is not possible. We can eliminate the above left recursion by rewriting the offending production as:
A → βA'
A' → αA' | epsilon

Pseudocode:

- Initially, Check if the given grammar contains left recursion, if present then separate the production and start working on it.
- Introduce a new non-terminal (append _DASH to the original non-terminal name)
- Append the newly created non-terminal to each terminal in the original production
- Write the newly created non-terminal in the LHS, and in the RHS write the following form: **n_DASH → n_DASH TERMINAL | epsilon**
- Repeat the previous step until all left recursion is eliminated from the grammar

(Elimination of Left Recursion)

6- Left Factoring:

Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracking by the parser.

Suppose an example: **A -> qB | qC**, where A, B, and C are non-terminals, and q is a terminal.

The grammar is converted into two productions:

A -> qD

D -> B|C

(Left factoring)

First Construction:

FIRST(A) is a set of the terminal symbols which occur as first symbols in strings derived from A where A is any string of grammar symbols. If A derives to epsilon, then FIRST(A) also contains epsilon.

Rules to compute FIRST set:
1- If x is a terminal, then FIRST(x) = {'x'}
2- If x-> Є, is a production rule, then add Є to FIRST(x).
3- If X->Y1 Y2 Y3…. Yn is a production, a. FIRST(X) = FIRST(Y1)
b. If FIRST(Y1) contains Є then FIRST(X) = {FIRST(Y1) – Є} U {FIRST(Y2)}
c. If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Pseudocode:

```
• Loop over the non-terminals, and for each non visited one:
    o Get first of production.
    o Append first into the non-terminal's entry in the first's table.
    o Mark non-terminal as visited.
• First recursive function
    o Loop over the right-hand side
        ▪ If terminal 'a' | eps
        ▪ First = First U {'a' | eps}.
    o If first is non-terminal A
        ▪ If non-terminal is not visited:
            • Recurse to find the firsts of A first.
            • Append first(A) to the entry.
    o If First table contains eps:
        ▪ Remove the epsilon and continue looping
    o Else break the loop
```

7- Follow Construction:

FOLLOW(A) is a defined to be the set of terminal symbols that appear immediately after the non terminal A in the right-hand side of any production.

During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Rules to compute FIRST set:

1- FOLLOW (Starting Symbol) = {$}

2- If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).

3- If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).

4- If A->pBq is a production and FIRST(q) contains Є, then FOLLOW(B) contains {FIRST(q) – Є} U FOLLOW(A)

Pseudocode:

```
• Loop over the non-terminals, and for each non visited one:
    o Loop over the right-hand side of this non-terminal's production.
        ▪ If current symbol is Nonterminal
            • If it's the last symbol in the right-hand side
                o Follow of the current depends on the follow
                  of the original non-terminal
            • Else
                o Loop over the rest of the right-hand side
                    ▪ If there's an epsilon in the start of the
                      current symbol
                        • Follow of the current depends
                          on the follow of the original
                          non-terminal
                    ▪ Follow of the current = first of the
                      next, except epsilon
        ▪ Else
            • Current symbol is a terminal, then add it to the fol-
              low of the original non-terminal
• While there are no more updates
    o Loop over all the follow dependencies
        ▪ Update the follow of each non-terminal
        ▪ If (old follow == new follow)
            • Break the loop
```

8- Parsing table construction:

It's a table made up of non-terminals and terminals that helps establish what grammar rule the parser should choose if it sees a non-terminal A on the top of its stack and a symbol a on its input stream.

We use the first and follow sets of the context free grammar to fill the parsing table.

Pseudocode:

```
• Loop over all the non-terminals
    o Loop over the first entries of each non-terminals
        ▪ If the current entry is epsilon
            • Boolean hasEpsilon = true
• Loop over all the productions (right-hand side)
    o For each terminal in the first
        ▪ Add an entry to the parsing table under the terminal
          column
    o If hasEpsilon is true
        ▪ For each terminal in the follow set
            • Add an entry to the parsing table under the
              terminal column
            • If current follow entry contains $
                o Boolean hasEndMarker = true
    o Else
        ▪ Add an entry with the SYNC_token
    o If hasEpsilon is true and hasEndMarker is true
        ▪ Add an entry $
• All the other entries in the parsing table are error entries
```

9- Parse:

Pseudocode:

```
• Create an empty stack
• Push the end marker and the starting symbol on the stack
• While stack isn't empty
    o If the top of the stack is a non-terminal
        ▪ If the top of the stack leads to an empty entry
            • Skip token (empty entry)
        ▪ Else if the top of the stack leads to an epsilon
            • Pop from the stack
        ▪ Else if the top of the stack leads to a SYNC_TOK
            • Pop from the stack
        ▪ Else /* Leads to a production rule */
            • Push the production rule to the STACK
    o Else /* Top of the stack is a terminal */
        ▪ If the input token matches the input of the stack
            • Print match action
        ▪ Else
            • Print no match, Skip the token
```

10- Backpatching:

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump

Back patching usually refers to the process of resolving forward branches that have been planted in the code, e.g. at 'if' statements, when the value of the target becomes known, e.g. when the closing brace or matching 'else' is encountered.

To manipulate lists of jumps, we use three functions:

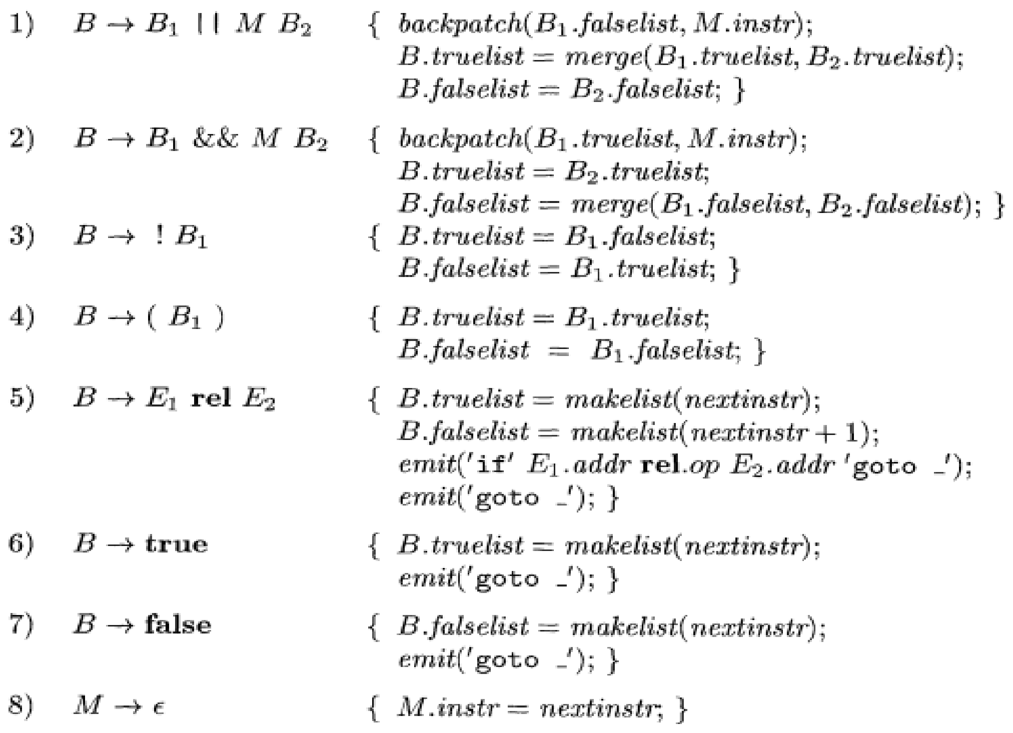
1. Make_list(i): creates a new list containing only i, an index into the array of instructions; make_list returns a pointer to the newly created list.

2. merge (pl, p2): concatenates the lists pointed to by pl and p2, and returns a pointer to the concatenated list.

3. backpatch (p, i): inserts i as the target label for each of the instructions on the list pointed to by p.

Backpatching for Boolean Expressions:

We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing. A marker nonterminal M in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated.

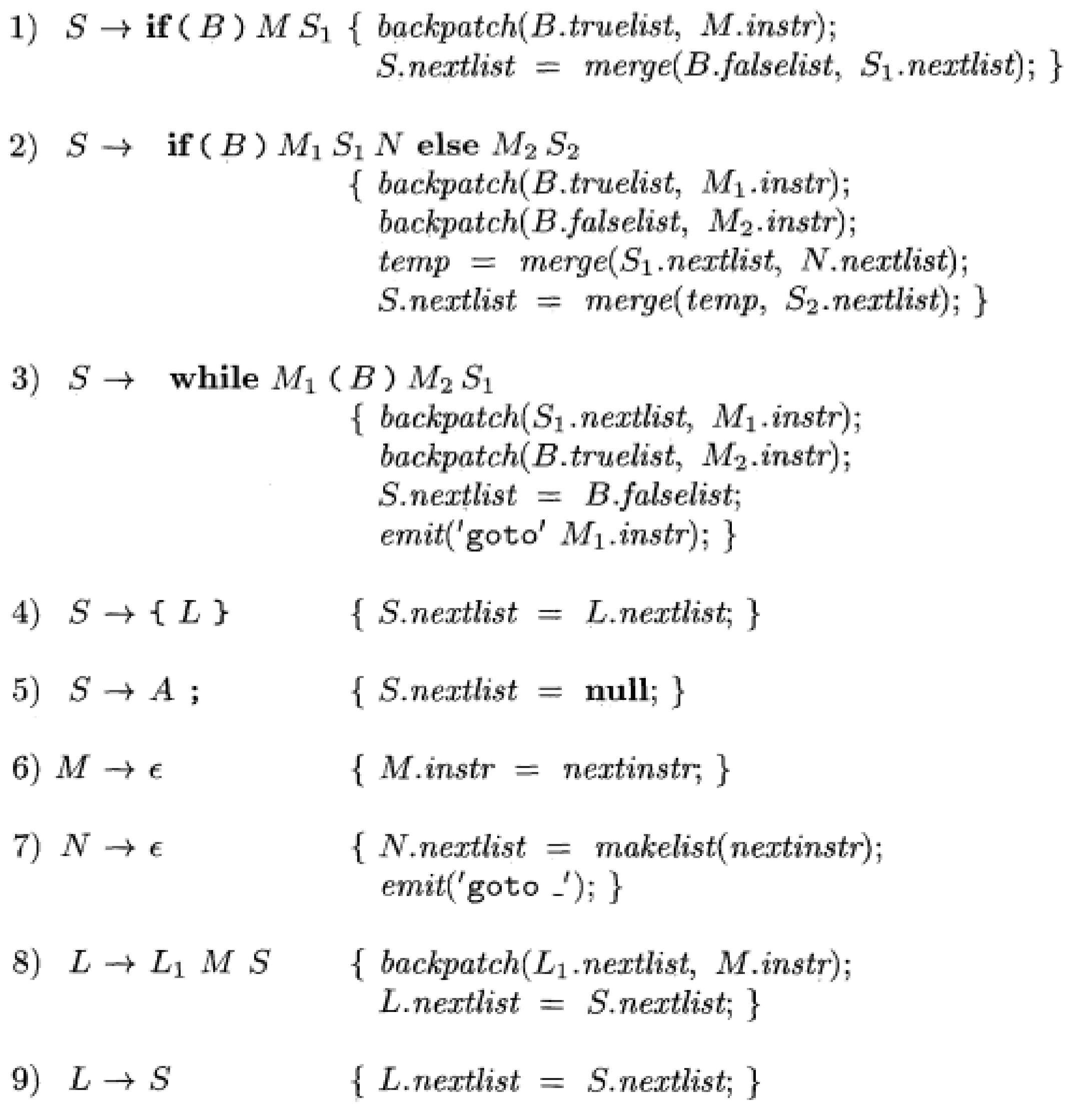


1)	$B \rightarrow B_1 \mid\mid M\ B_2$	$\{$ $backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;$ $\}$
2)	$B \rightarrow B_1\ \&\&\ M\ B_2$	$\{$ $backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);$ $\}$
3)	$B \rightarrow\ !\ B_1$	$\{$ $B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;$ $\}$
4)	$B \rightarrow (\ B_1\ )$	$\{$ $B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist;$ $\}$
5)	$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$\{$ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $emit($'if' $E_1.addr\ \mathbf{rel}.op\ E_2.addr$ 'goto _'$);$ $emit($'goto _'$);$ $\}$
6)	$B \rightarrow \mathbf{true}$	$\{$ $B.truelist = makelist(nextinstr);$ $emit($'goto _'$);$ $\}$
7)	$B \rightarrow \mathbf{false}$	$\{$ $B.falselist = makelist(nextinstr);$ $emit($'goto _'$);$ $\}$
8)	$M \rightarrow \epsilon$	$\{$ $M.instr = nextinstr;$ $\}$

Figure 4: Translation scheme for boolean expressions

Flow-of-control statements:

Here S denotes a statement, L a statement list, A an assignment-statement, and B a boolean expression.



1) $S \rightarrow \textbf{if} (B) \; M \; S_1$	$\{ \; backpatch(B.truelist, \; M.instr);$ $S.nextlist \; = \; merge(B.falselist, \; S_1.nextlist); \; \}$
2) $S \rightarrow \textbf{if} (B) \; M_1 \; S_1 \; N \; \textbf{else} \; M_2 \; S_2$	$\{ \; backpatch(B.truelist, \; M_1.instr);$ $backpatch(B.falselist, \; M_2.instr);$ $temp \; = \; merge(S_1.nextlist, \; N.nextlist);$ $S.nextlist \; = \; merge(temp, \; S_2.nextlist); \; \}$
3) $S \rightarrow \textbf{while} \; M_1 \; (B) \; M_2 \; S_1$	$\{ \; backpatch(S_1.nextlist, \; M_1.instr);$ $backpatch(B.truelist, \; M_2.instr);$ $S.nextlist \; = \; B.falselist;$ $emit(\text{'goto'} \; M_1.instr); \; \}$
4) $S \rightarrow \{ \; L \; \}$	$\{ \; S.nextlist \; = \; L.nextlist; \; \}$
5) $S \rightarrow A \; ;$	$\{ \; S.nextlist \; = \; \textbf{null}; \; \}$
6) $M \rightarrow \epsilon$	$\{ \; M.instr \; = \; nextinstr; \; \}$
7) $N \rightarrow \epsilon$	$\{ \; N.nextlist \; = \; makelist(nextinstr);$ $emit(\text{'goto \_'}); \; \}$
8) $L \rightarrow L_1 \; M \; S$	$\{ \; backpatch(L_1.nextlist, \; M.instr);$ $L.nextlist \; = \; S.nextlist; \; \}$
9) $L \rightarrow S$	$\{ \; L.nextlist \; = \; S.nextlist; \; \}$

Figure 5: Translation of statements

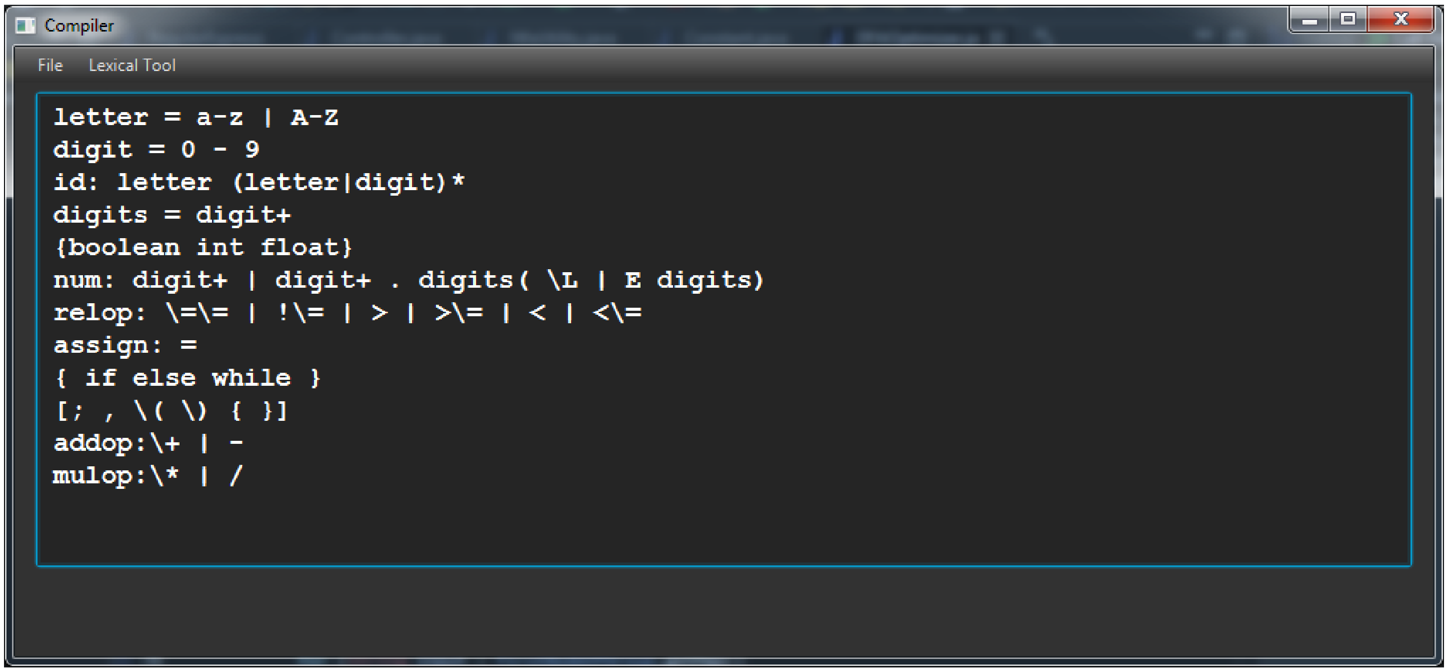
(Compilers Principles, Techniques, and Tools, 2006)

Sample runs:

1- Lexical Analyzer:

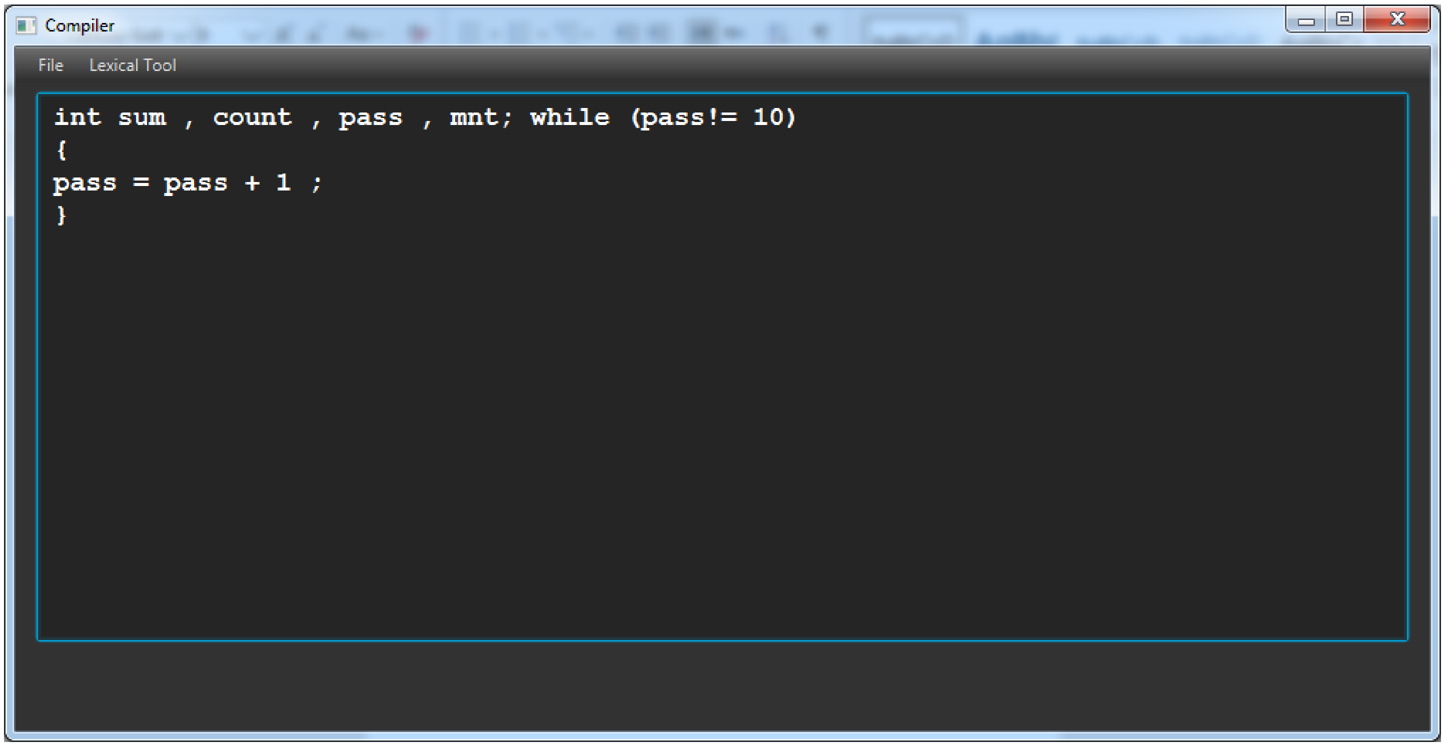
Test case 1:

Language Rules:



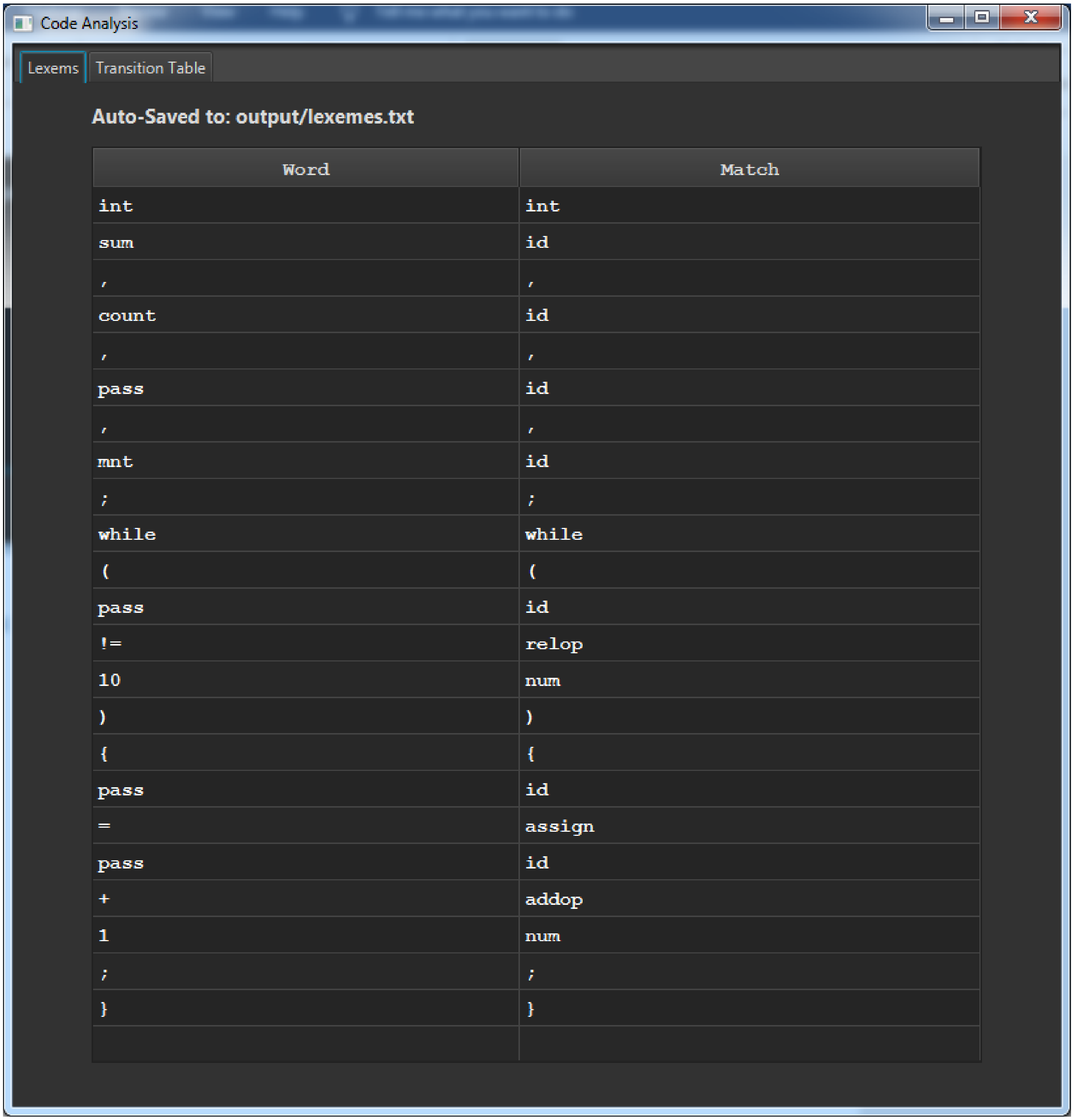
```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: =
{ if else while }
[; , \( \) { }]
addop:\+ | -
mulop:\* | /
```

Program:



```
int sum , count , pass , mnt; while (pass!= 10)
{
pass = pass + 1 ;
}
```

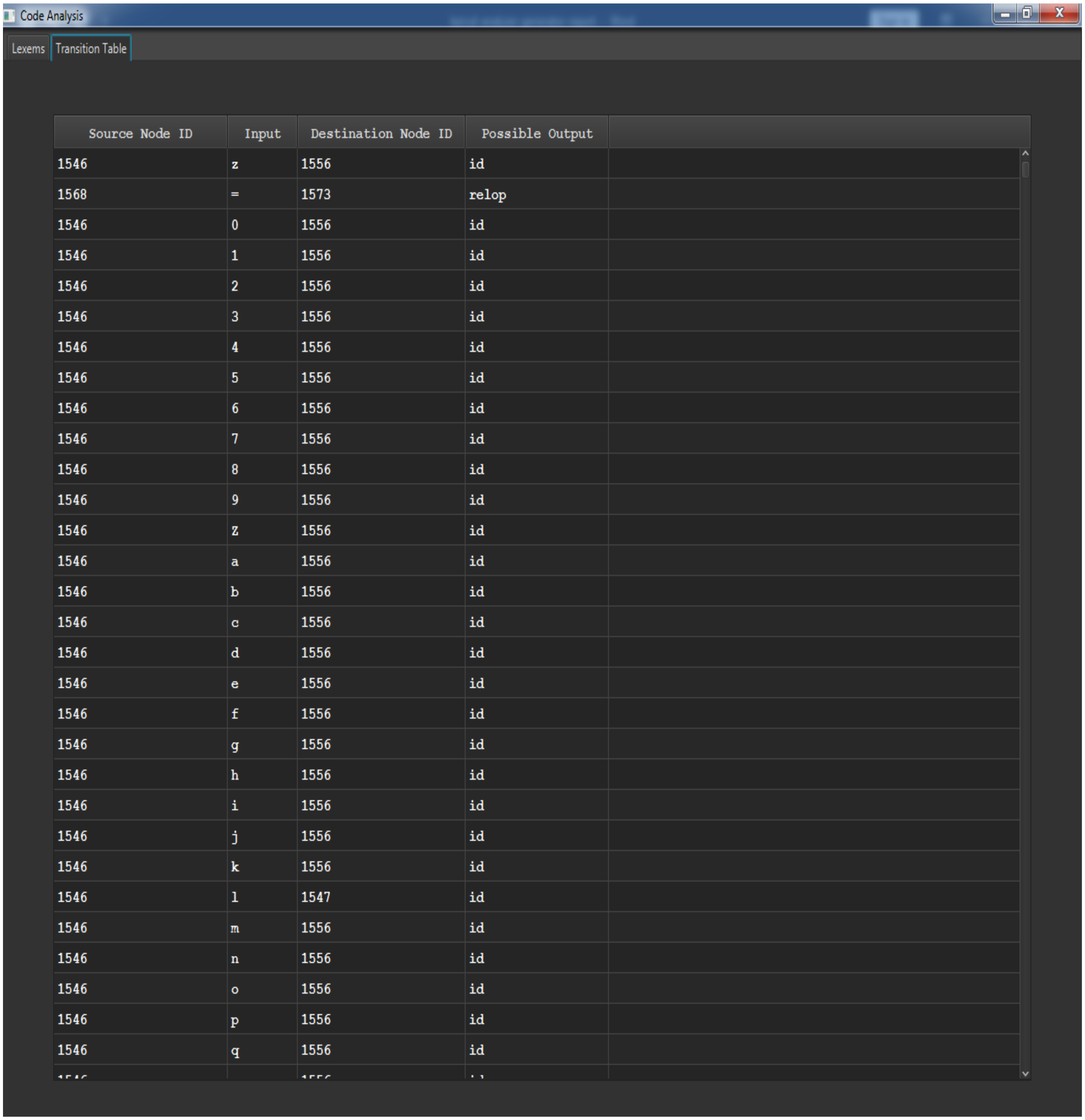
Lexemes:



Auto-Saved to: output/lexemes.txt

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
;	;
}	}

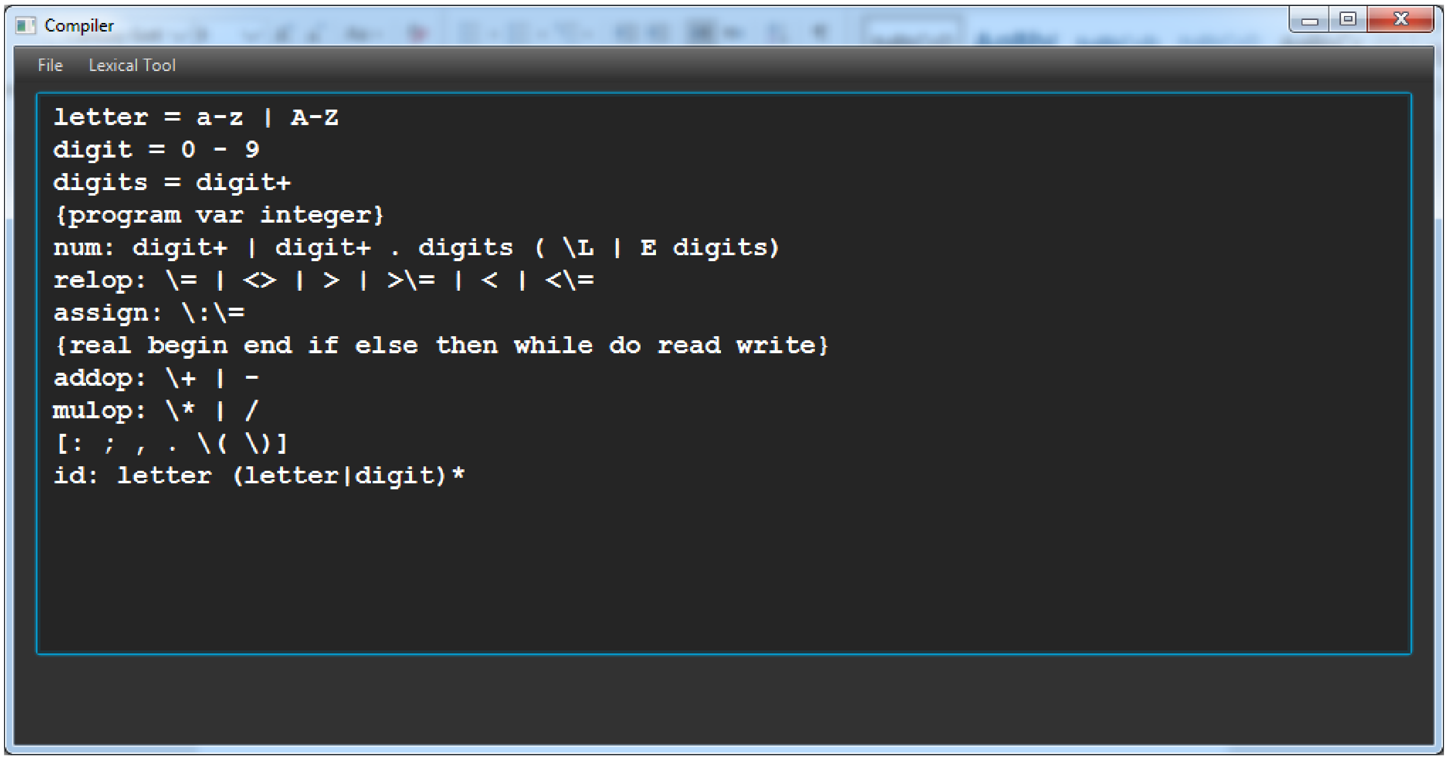
Part of the transition table of the dfa [Test case 1]:



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

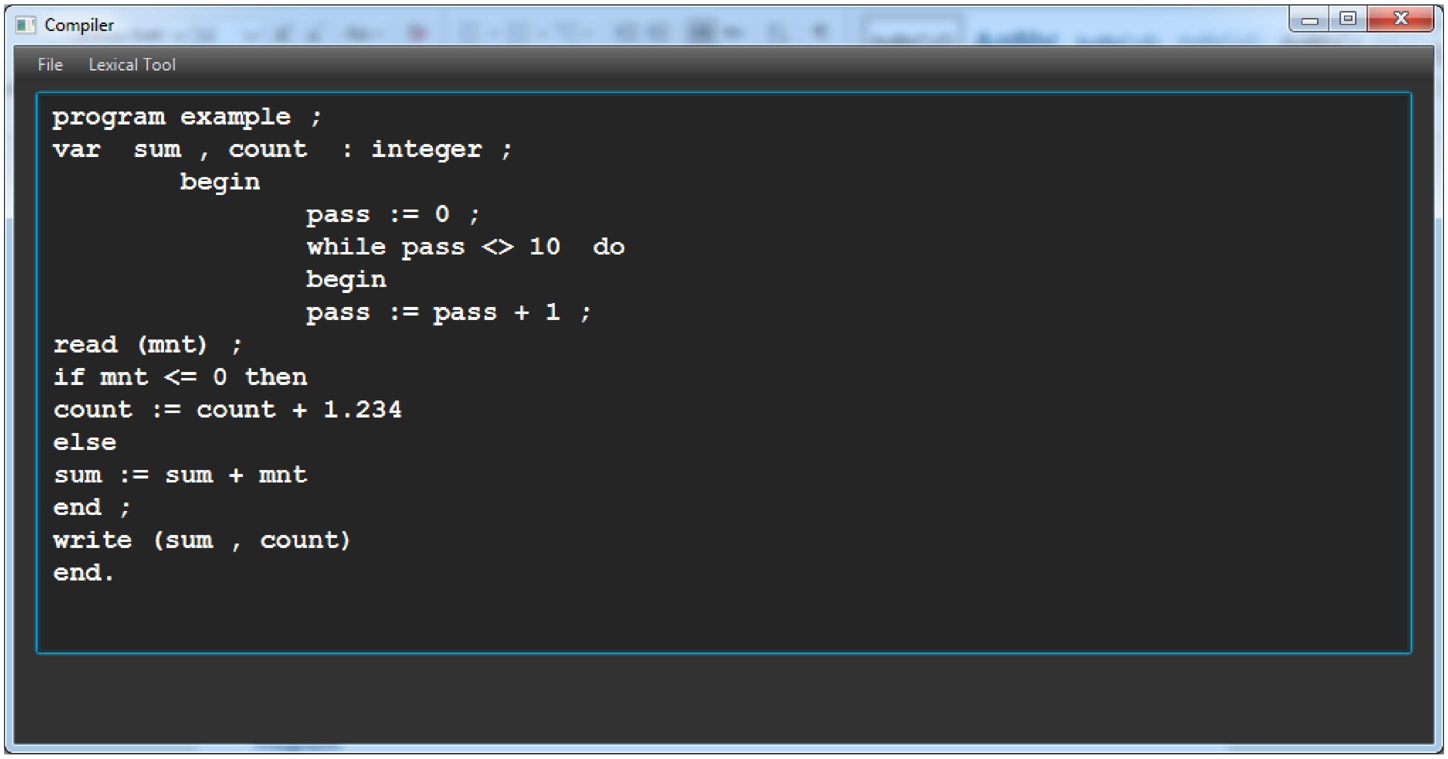
Test case 2:

Language rules:



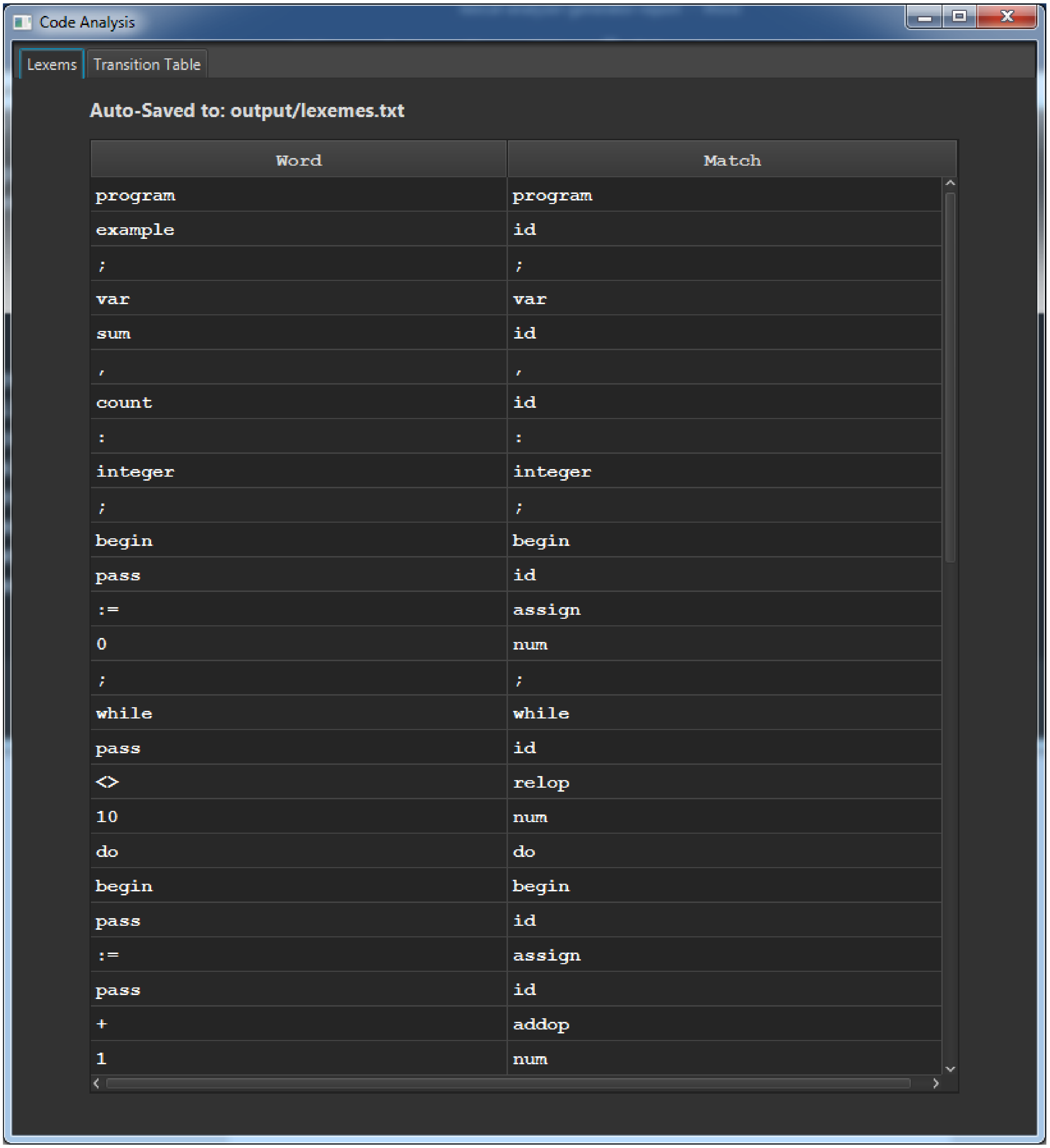
```
letter = a-z | A-Z
digit = 0 - 9
digits = digit+
{program var integer}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \= | <> | > | >\= | < | <\=
assign: \:\=
{real begin end if else then while do read write}
addop: \+ | -
mulop: \* | /
[: ; , . \( \)]
id: letter (letter|digit)*
```

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.
```

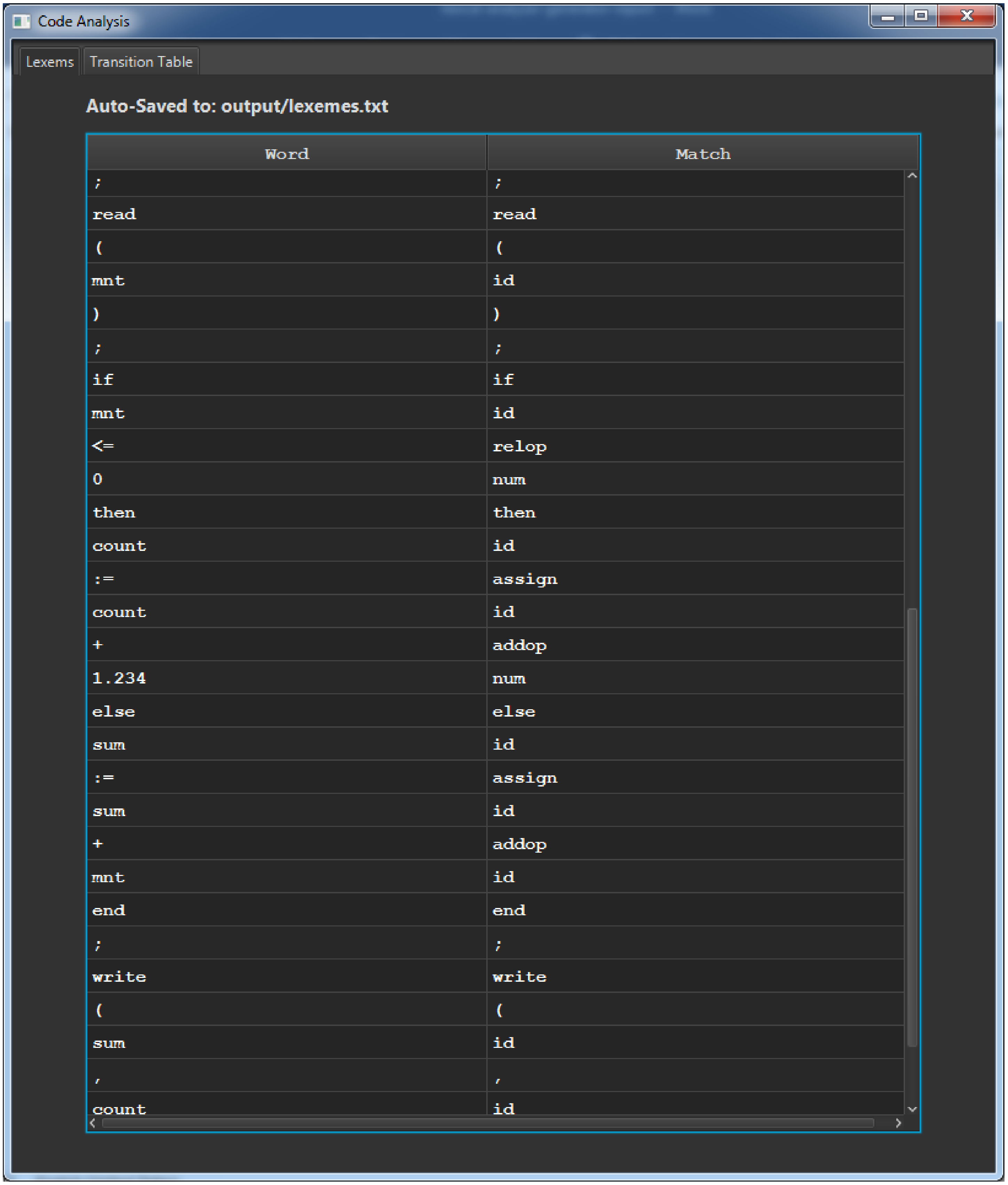
Lexemes:



Auto-Saved to: output/lexemes.txt

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

Rest of the lexemes:



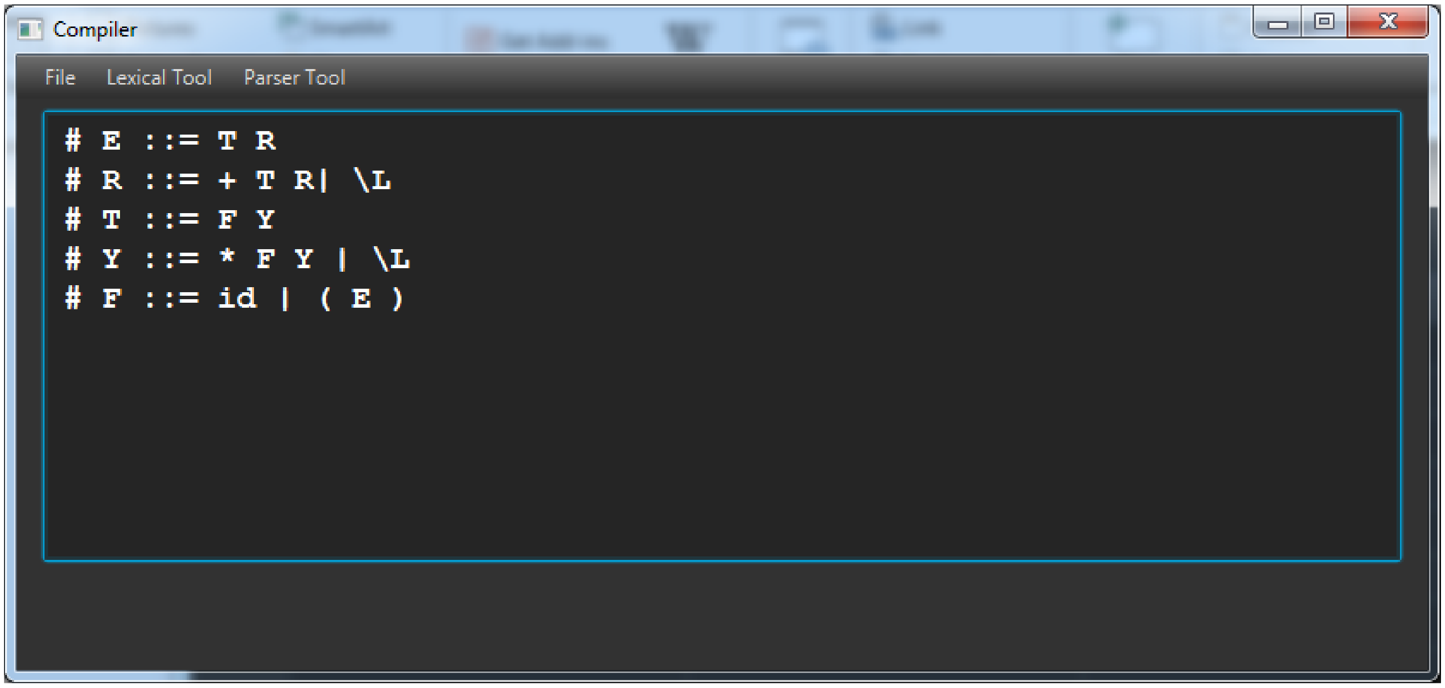
Auto-Saved to: output/lexemes.txt

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

2- Parser generator:

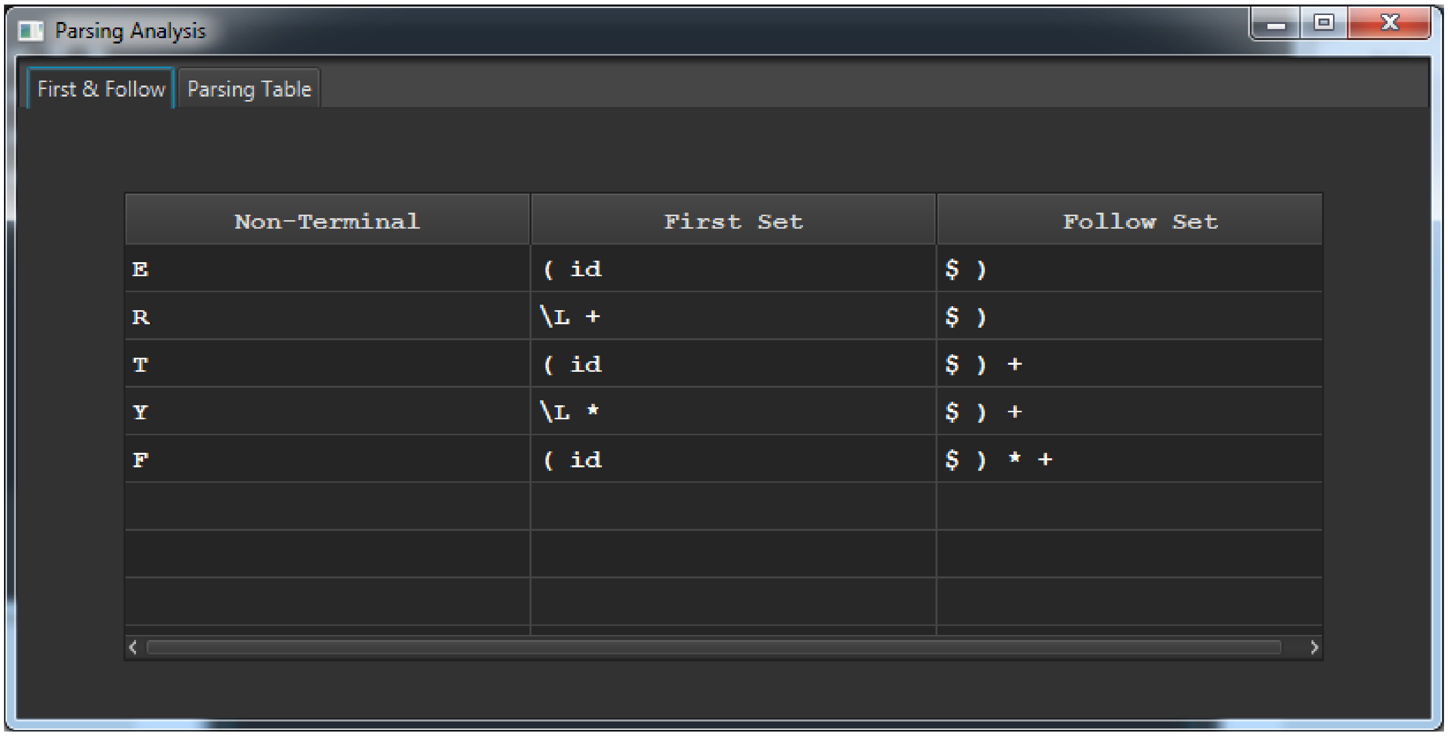
Test case 1:

Context free grammar:



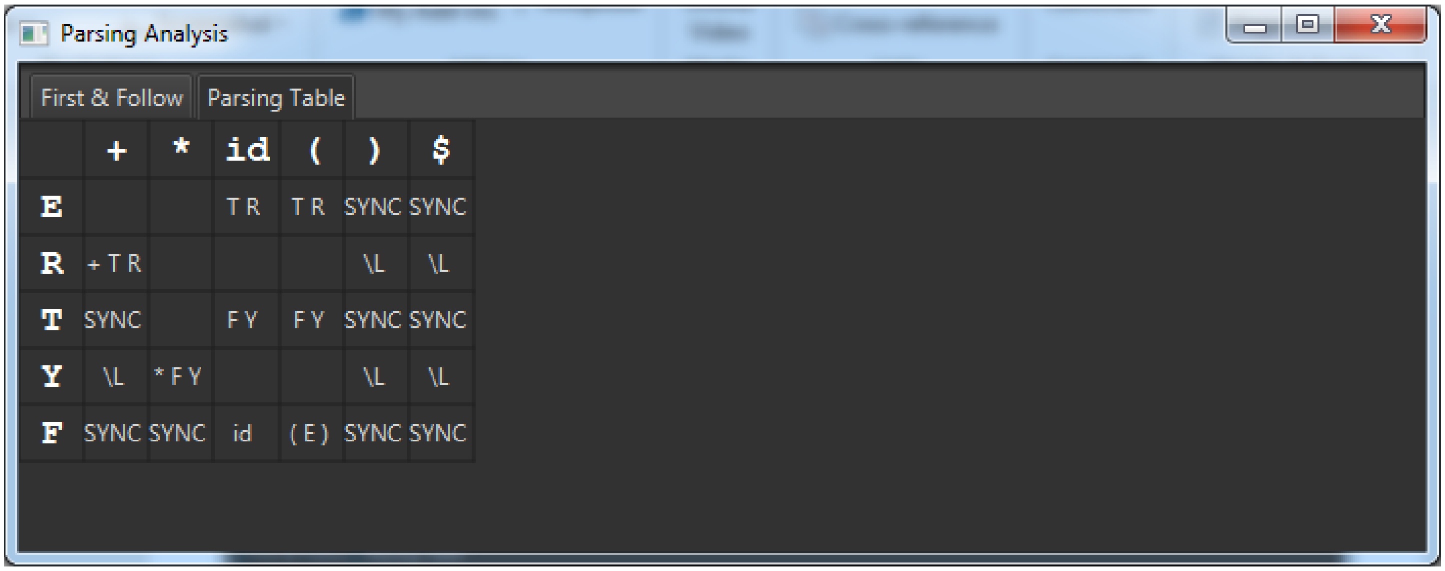
```
# E ::= T R
# R ::= + T R| \L
# T ::= F Y
# Y ::= * F Y | \L
# F ::= id | ( E )
```

First and follow sets:



Non-Terminal	First Set	Follow Set
E	(id	$)
R	\L +	$)
T	(id	$) +
Y	\L *	$) +
F	(id	$) * +

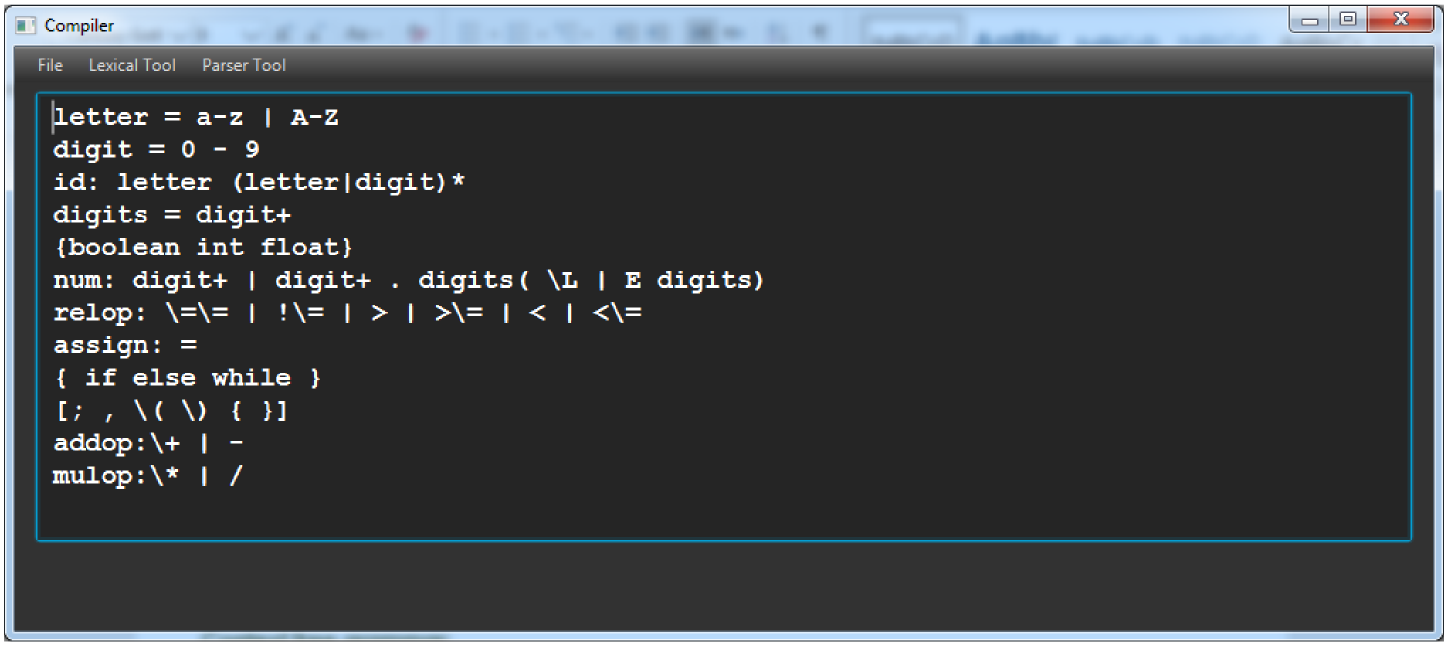
Parsing table:



	+	*	id	(	)	$
E			T R	T R	SYNC	SYNC
R	+ T R				\L	\L
T	SYNC		F Y	F Y	SYNC	SYNC
Y	\L	* F Y			\L	\L
F	SYNC	SYNC	id	(E)	SYNC	SYNC

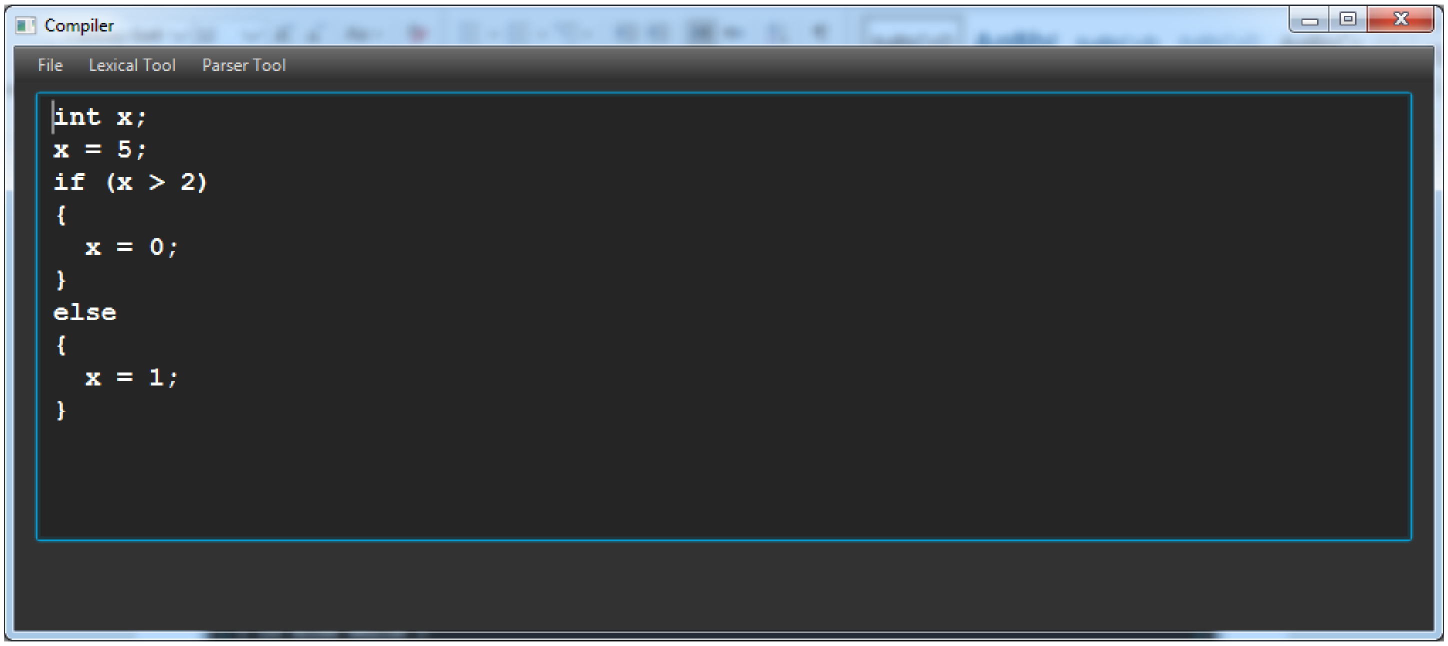
Test case 2:

Lexical rules:



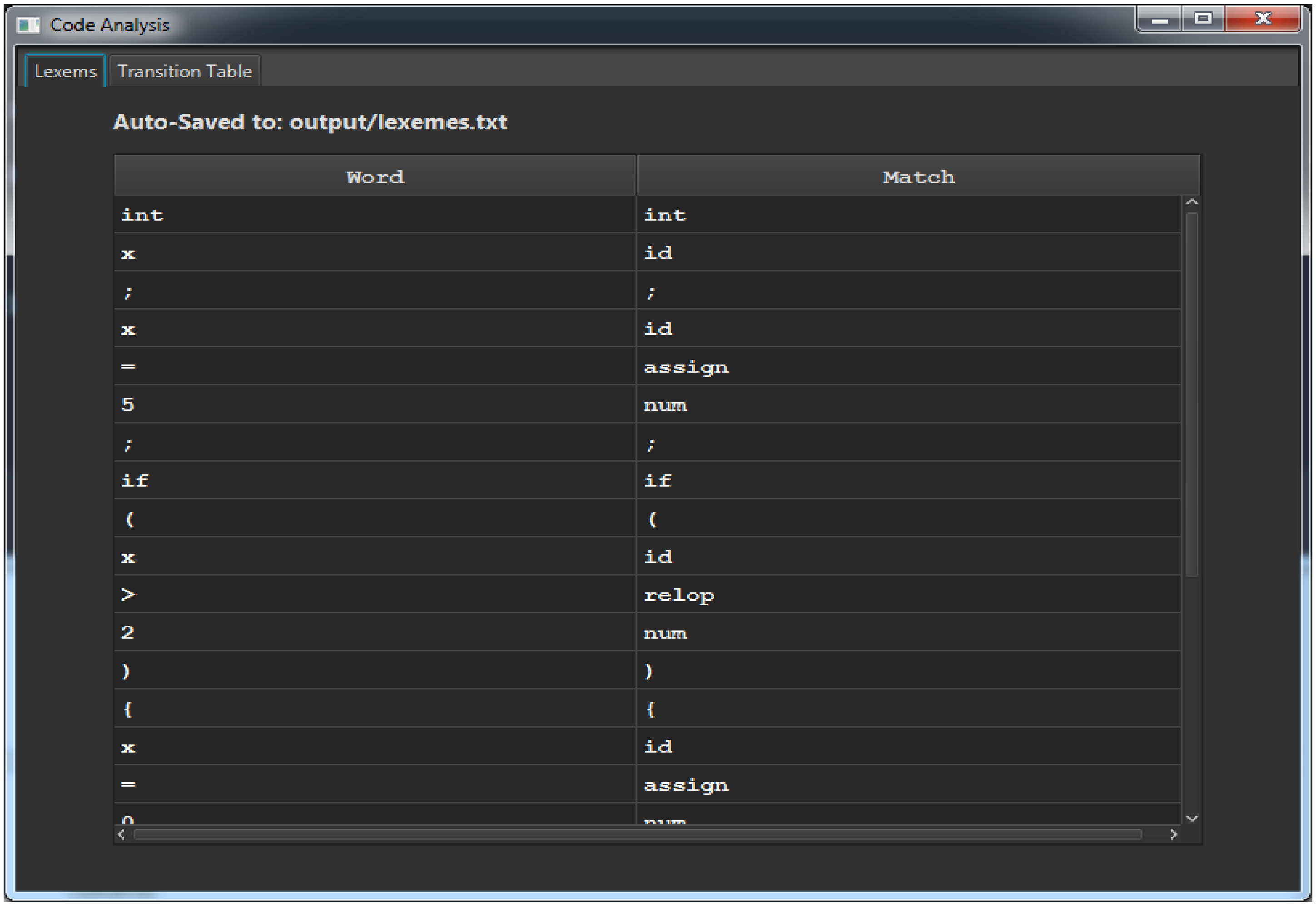
```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: =
{ if else while }
[; , \( \) { }]
addop:\+ | -
mulop:\* | /
```

Programs:



```
int x;
x = 5;
if (x > 2)
{
  x = 0;
}
else
{
  x = 1;
}
```

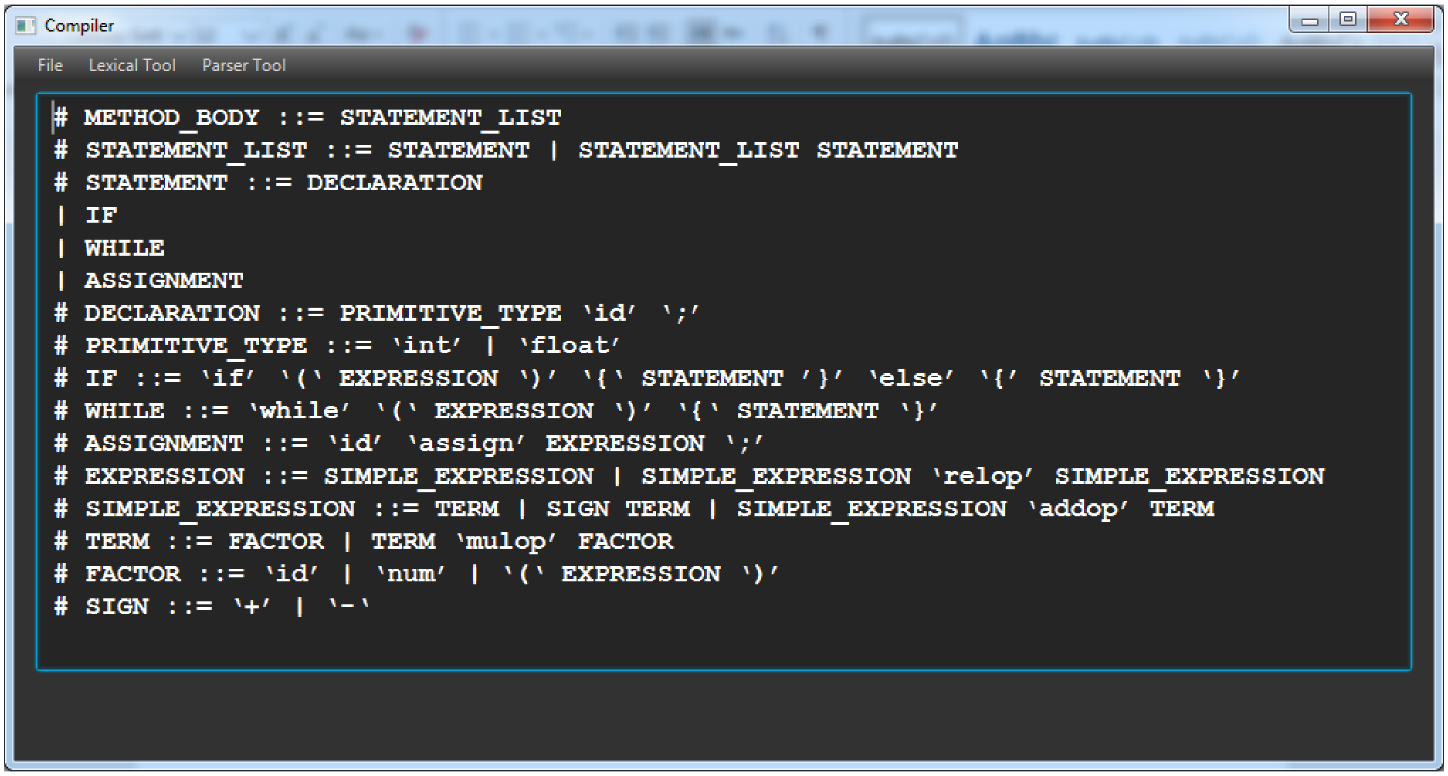
Lexemes:



Auto-Saved to: output/lexemes.txt

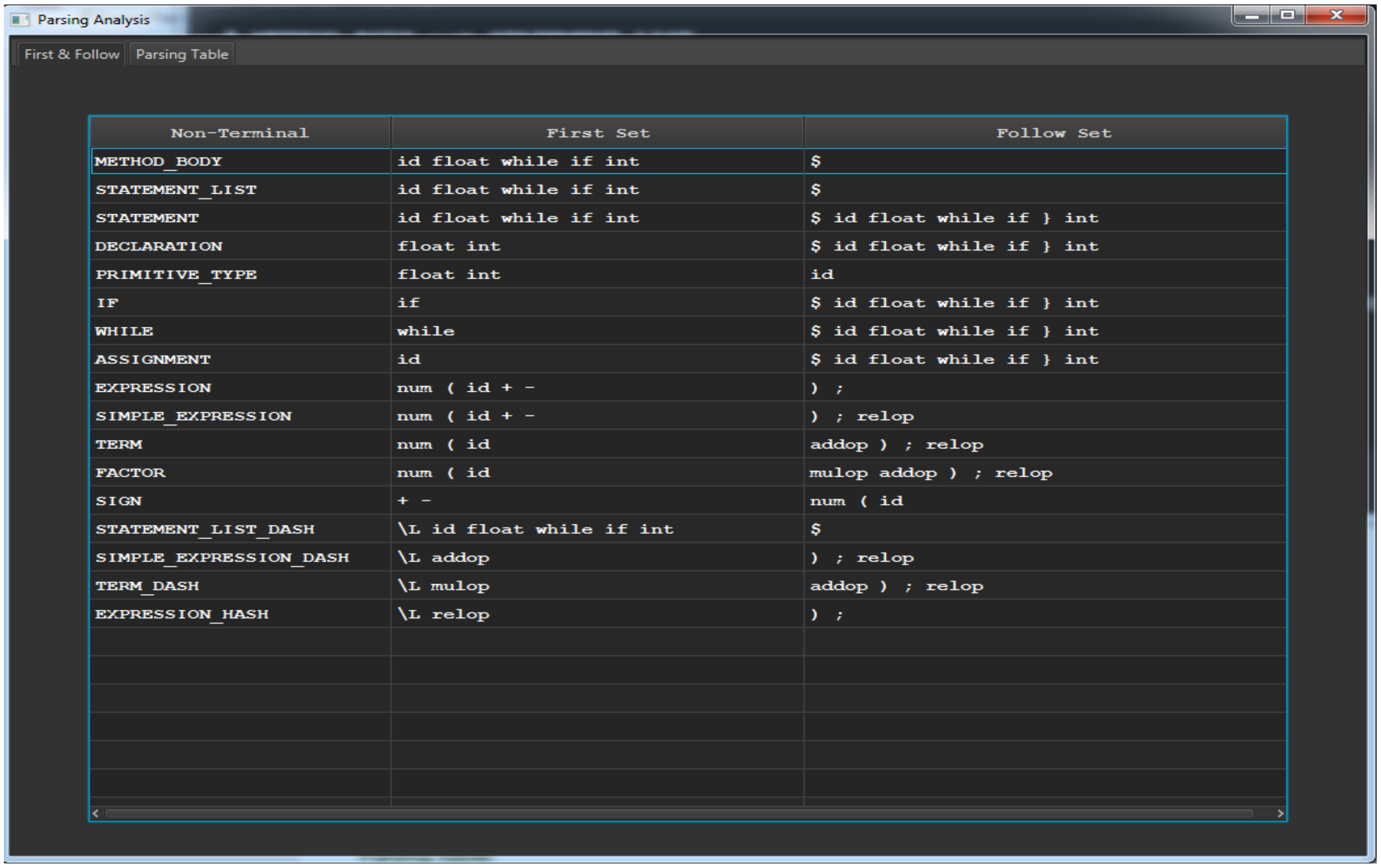
Word	Match
int	int
x	id
;	;
x	id
=	assign
5	num
;	;
if	if
(	(
x	id
>	relop
2	num
)	)
{	{
x	id
=	assign
0	num

Context free grammar:



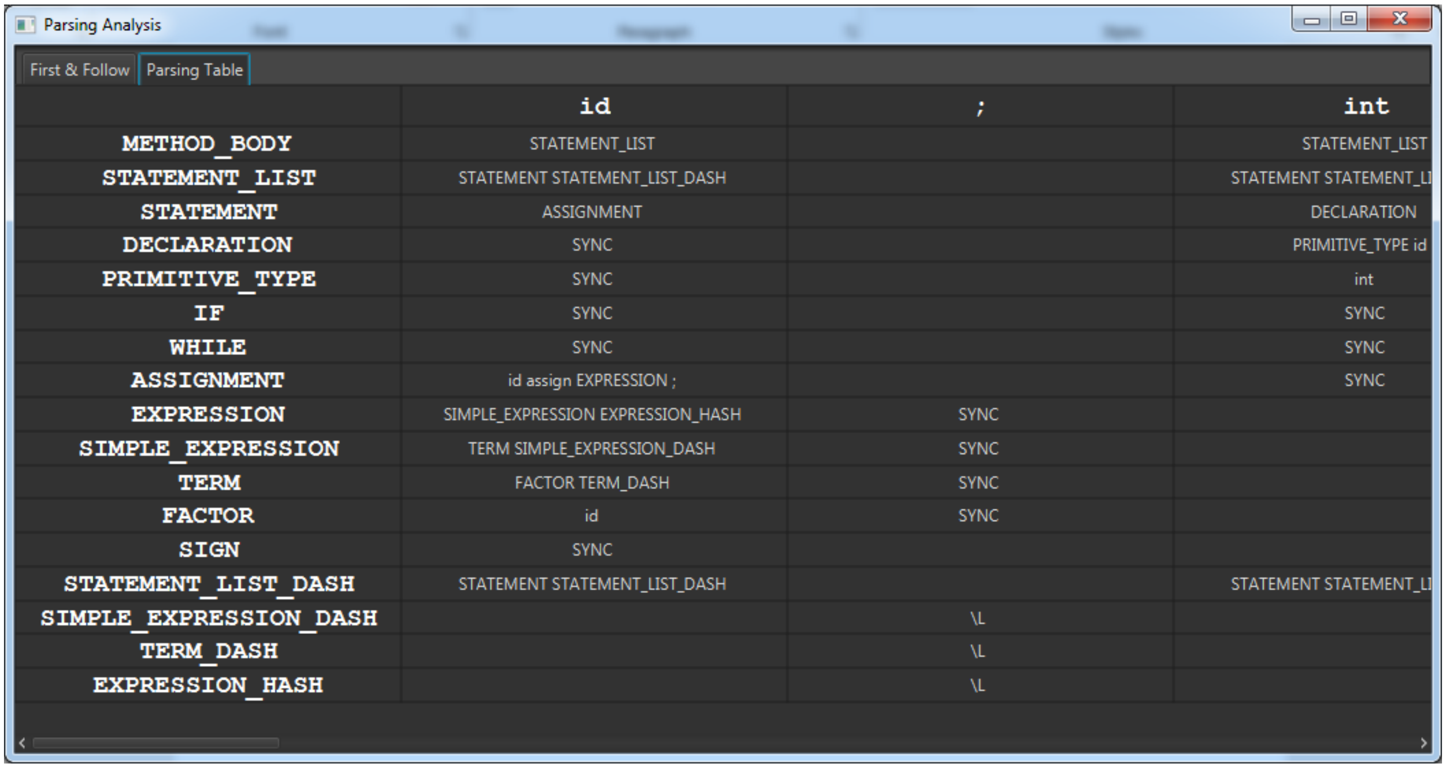
```
# METHOD_BODY ::= STATEMENT_LIST
# STATEMENT_LIST ::= STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT ::= DECLARATION
| IF
| WHILE
| ASSIGNMENT
# DECLARATION ::= PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE ::= 'int' | 'float'
# IF ::= 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
# 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 ::= '+' | '-'
```

First and follow sets:



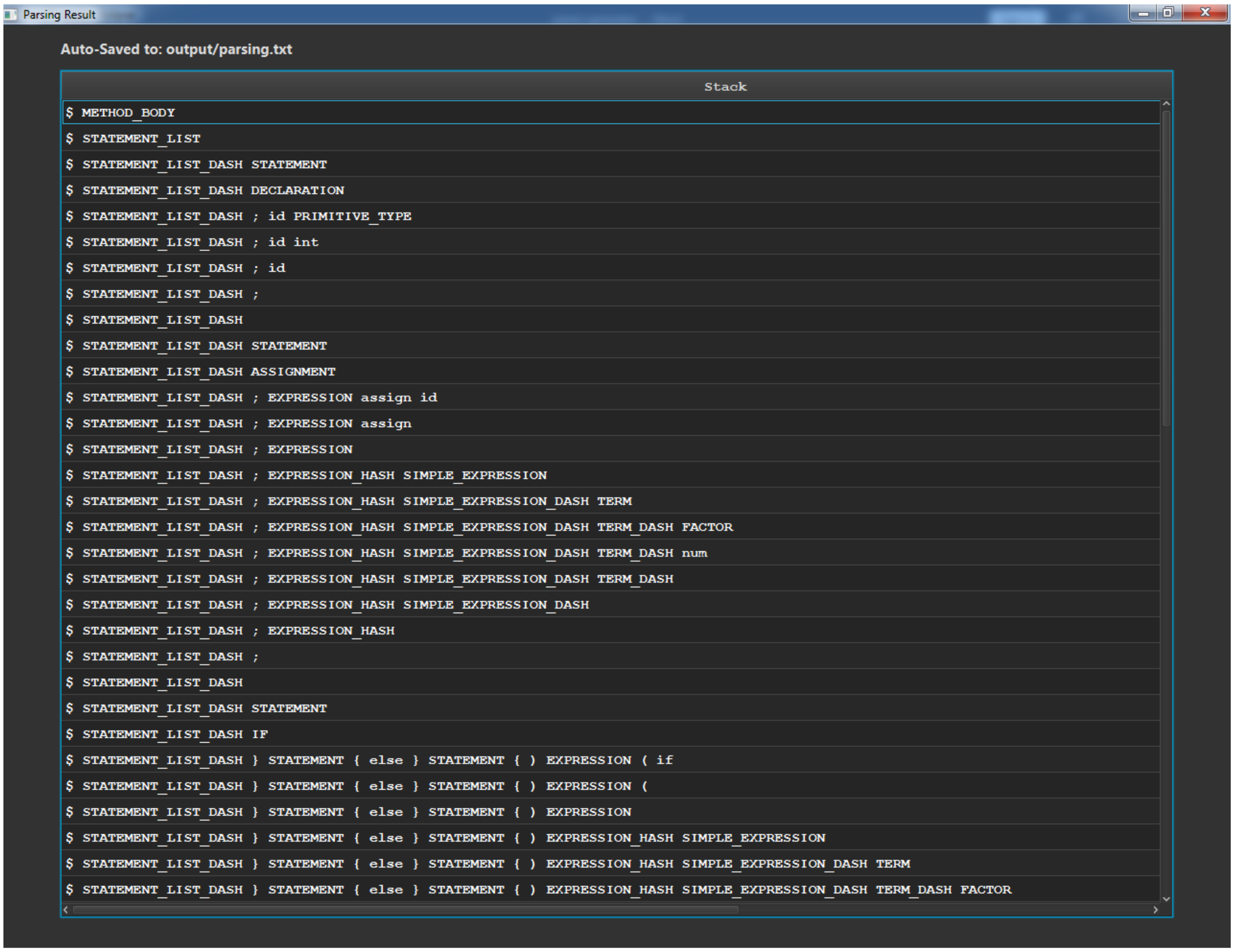
Non-Terminal	First Set	Follow Set
METHOD_BODY	id float while if int	$
STATEMENT_LIST	id float while if int	$
STATEMENT	id float while if int	$ id float while if } int
DECLARATION	float int	$ id float while if } int
PRIMITIVE_TYPE	float int	id
IF	if	$ id float while if } int
WHILE	while	$ id float while if } int
ASSIGNMENT	id	$ id float while if } int
EXPRESSION	num (id + -	) ;
SIMPLE_EXPRESSION	num (id + -	) ; relop
TERM	num (id	addop) ; relop
FACTOR	num (id	mulop addop) ; relop
SIGN	+ -	num (id
STATEMENT_LIST_DASH	\L id float while if int	$
SIMPLE_EXPRESSION_DASH	\L addop	) ; relop
TERM_DASH	\L mulop	addop) ; relop
EXPRESSION_HASH	\L relop	) ;

Parsing table:



	id	;	int
METHOD_BODY	STATEMENT_LIST		STATEMENT_LIST
STATEMENT_LIST	STATEMENT STATEMENT_LIST_DASH		STATEMENT STATEMENT_LI
STATEMENT	ASSIGNMENT		DECLARATION
DECLARATION	SYNC		PRIMITIVE_TYPE id
PRIMITIVE_TYPE	SYNC		int
IF	SYNC		SYNC
WHILE	SYNC		SYNC
ASSIGNMENT	id assign EXPRESSION ;		SYNC
EXPRESSION	SIMPLE_EXPRESSION EXPRESSION_HASH	SYNC	
SIMPLE_EXPRESSION	TERM SIMPLE_EXPRESSION_DASH	SYNC	
TERM	FACTOR TERM_DASH	SYNC	
FACTOR	id	SYNC	
SIGN	SYNC		
STATEMENT_LIST_DASH	STATEMENT STATEMENT_LIST_DASH		STATEMENT STATEMENT_LI
SIMPLE_EXPRESSION_DASH		\L	
TERM_DASH		\L	
EXPRESSION_HASH		\L	

Parser output:



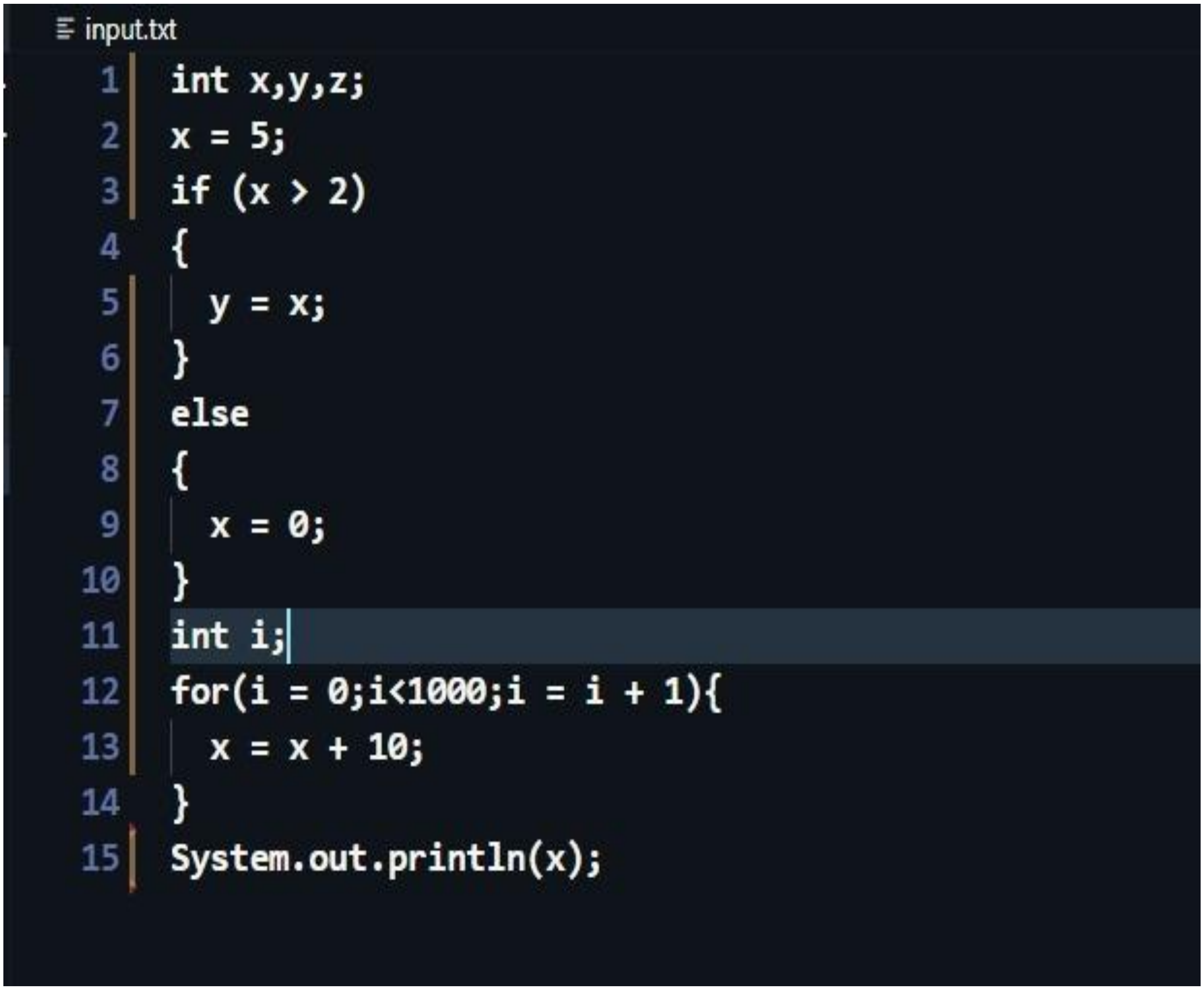
Auto-Saved to: output/parsing.txt

```
Stack
$ METHOD_BODY
$ STATEMENT_LIST
$ STATEMENT_LIST_DASH STATEMENT
$ STATEMENT_LIST_DASH DECLARATION
$ STATEMENT_LIST_DASH ; id PRIMITIVE_TYPE
$ STATEMENT_LIST_DASH ; id int
$ STATEMENT_LIST_DASH ; id
$ STATEMENT_LIST_DASH ;
$ STATEMENT_LIST_DASH
$ STATEMENT_LIST_DASH STATEMENT
$ STATEMENT_LIST_DASH ASSIGNMENT
$ STATEMENT_LIST_DASH ; EXPRESSION assign id
$ STATEMENT_LIST_DASH ; EXPRESSION assign
$ STATEMENT_LIST_DASH ; EXPRESSION
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH FACTOR
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH num
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH
$ STATEMENT_LIST_DASH ;
$ STATEMENT_LIST_DASH
$ STATEMENT_LIST_DASH STATEMENT
$ STATEMENT_LIST_DASH IF
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT { ) EXPRESSION ( if
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT { ) EXPRESSION (
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT { ) EXPRESSION
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT { ) EXPRESSION_HASH SIMPLE_EXPRESSION
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT { ) EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT { ) EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH FACTOR
```

3- Code generation:

Test case 1 (Working program):

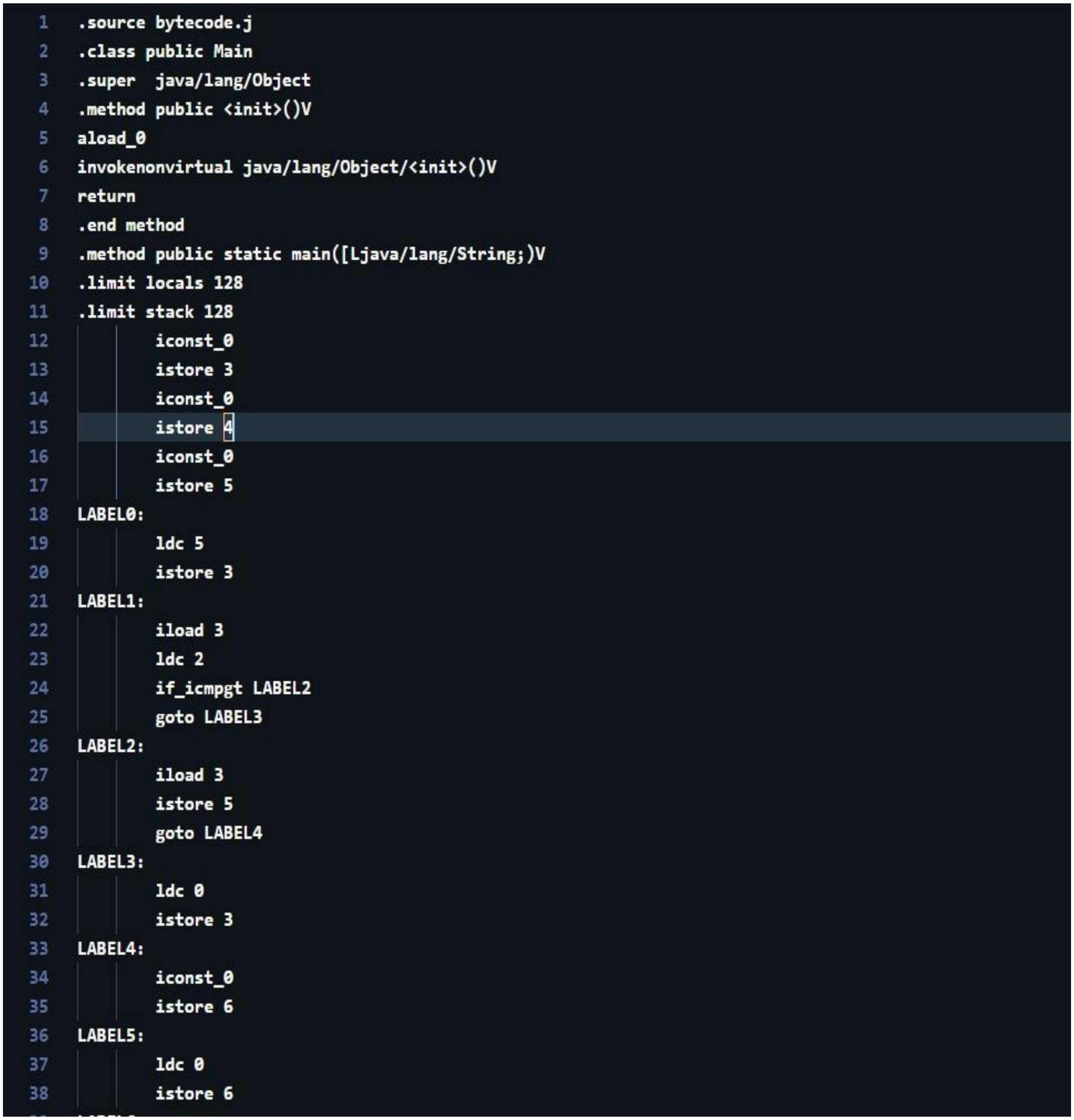
Program:



input.txt

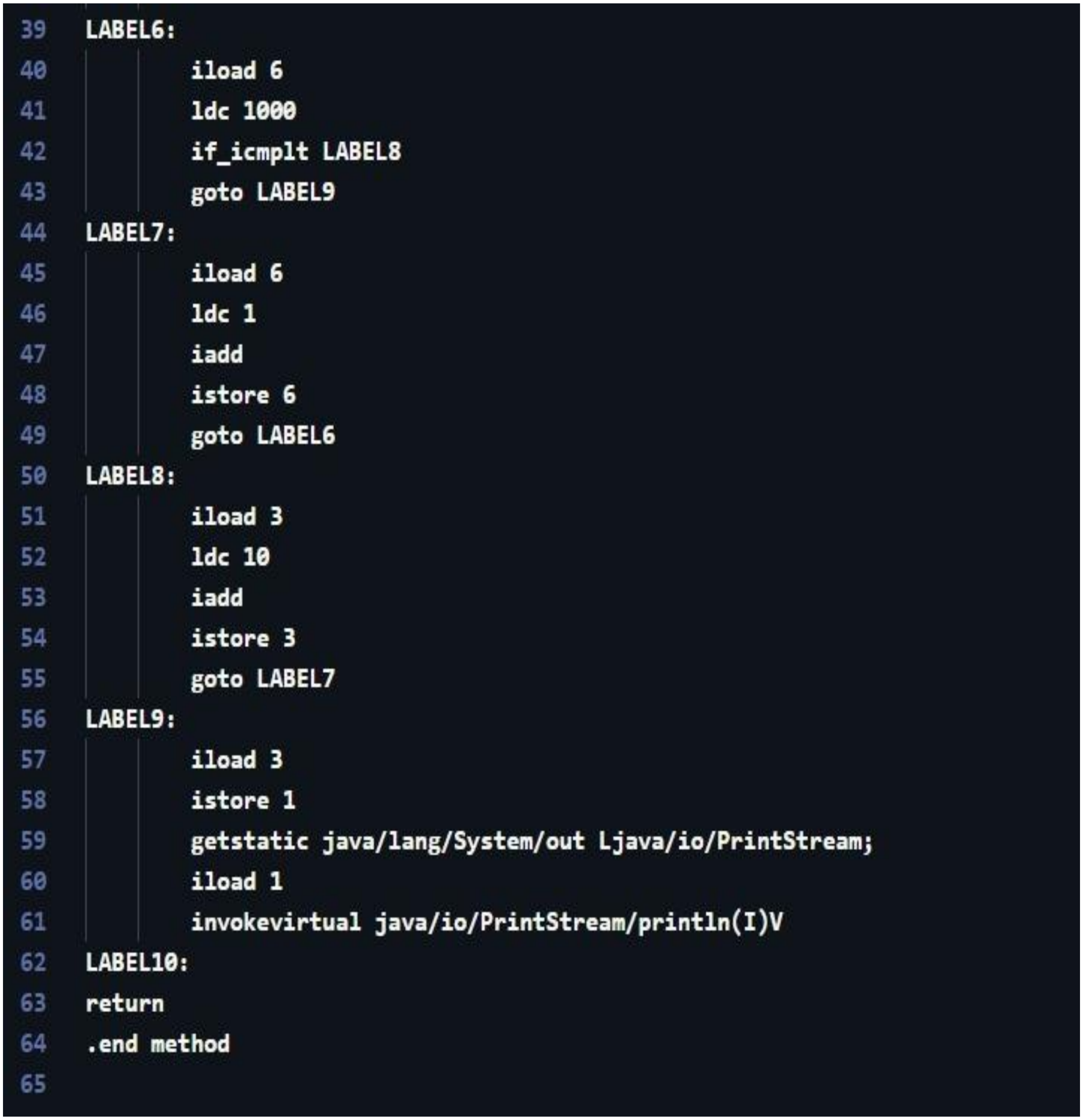
```
int x,y,z;
x = 5;
if (x > 2)
{
  y = x;
}
else
{
  x = 0;
}
int i;
for(i = 0;i<1000;i = i + 1){
  x = x + 10;
}
System.out.println(x);
```

Resultant Java bytecode:



```
.source bytecode.j
.class public Main
.super  java/lang/Object
.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return
.end method
.method public static main([Ljava/lang/String;)V
.limit locals 128
.limit stack 128
        iconst_0
        istore 3
        iconst_0
        istore 4
        iconst_0
        istore 5
LABEL0:
        ldc 5
        istore 3
LABEL1:
        iload 3
        ldc 2
        if_icmpgt LABEL2
        goto LABEL3
LABEL2:
        iload 3
        istore 5
        goto LABEL4
LABEL3:
        ldc 0
        istore 3
LABEL4:
        iconst_0
        istore 6
LABEL5:
        ldc 0
        istore 6
```

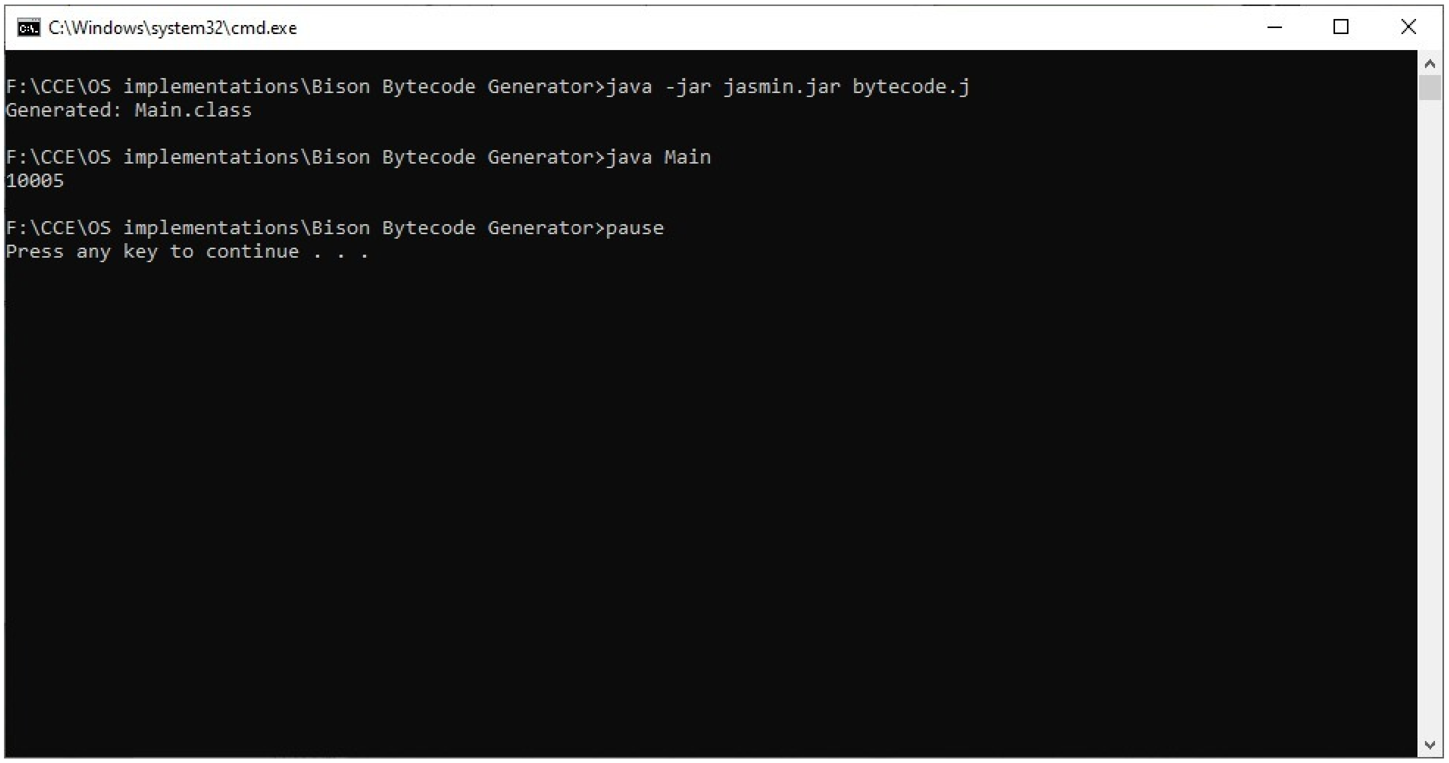
Rest of the Java bytecode:



```
LABEL6:
        iload 6
        ldc 1000
        if_icmplt LABEL8
        goto LABEL9
LABEL7:
        iload 6
        ldc 1
        iadd
        istore 6
        goto LABEL6
LABEL8:
        iload 3
        ldc 10
        iadd
        istore 3
        goto LABEL7
LABEL9:
        iload 3
        istore 1
        getstatic java/lang/System/out Ljava/io/PrintStream;
        iload 1
        invokevirtual java/io/PrintStream/println(I)V
LABEL10:
return
.end method
```

Jasmin Result:

Expected output is the final value of x (10005)



C:\Windows\system32\cmd.exe

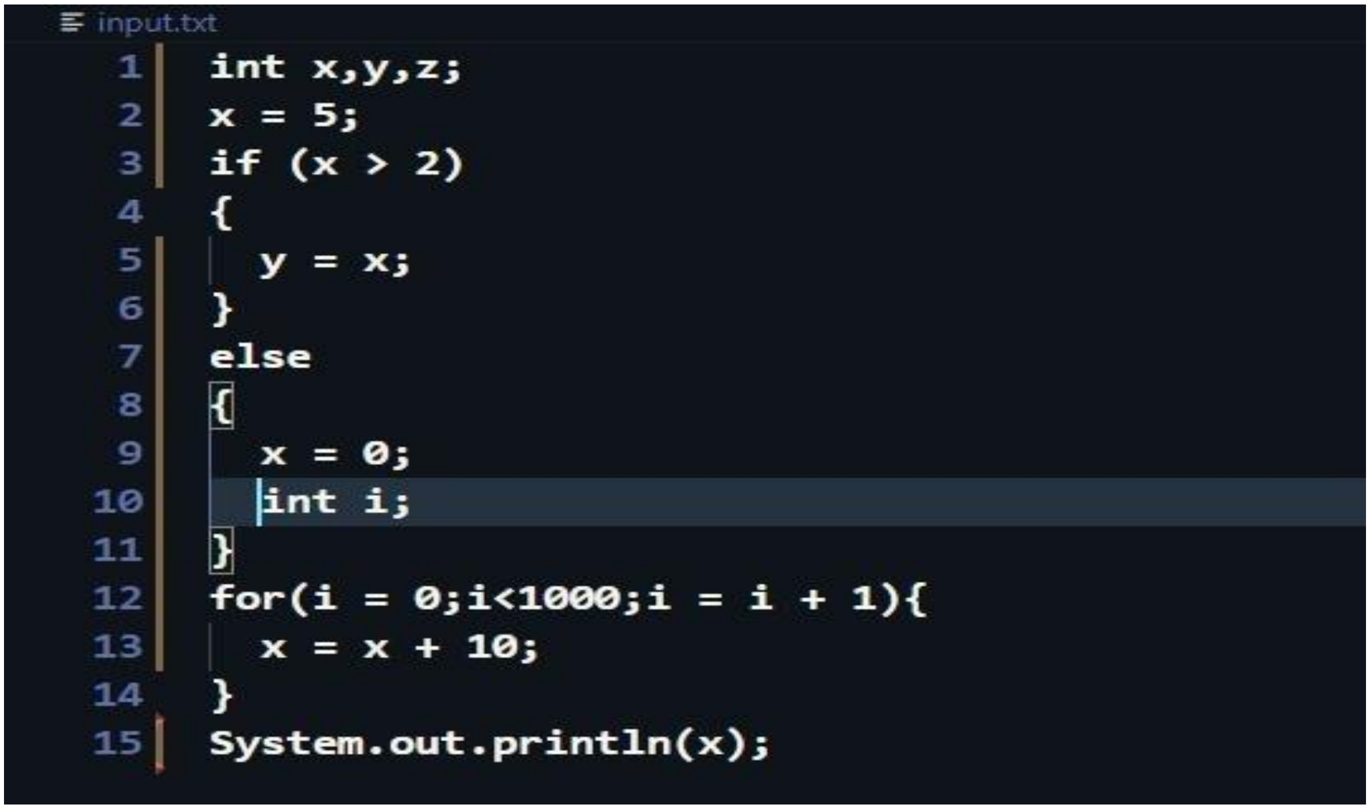
```
F:\CCE\OS implementations\Bison Bytecode Generator>java -jar jasmin.jar bytecode.j
Generated: Main.class

F:\CCE\OS implementations\Bison Bytecode Generator>java Main
10005

F:\CCE\OS implementations\Bison Bytecode Generator>pause
Press any key to continue . . .
```

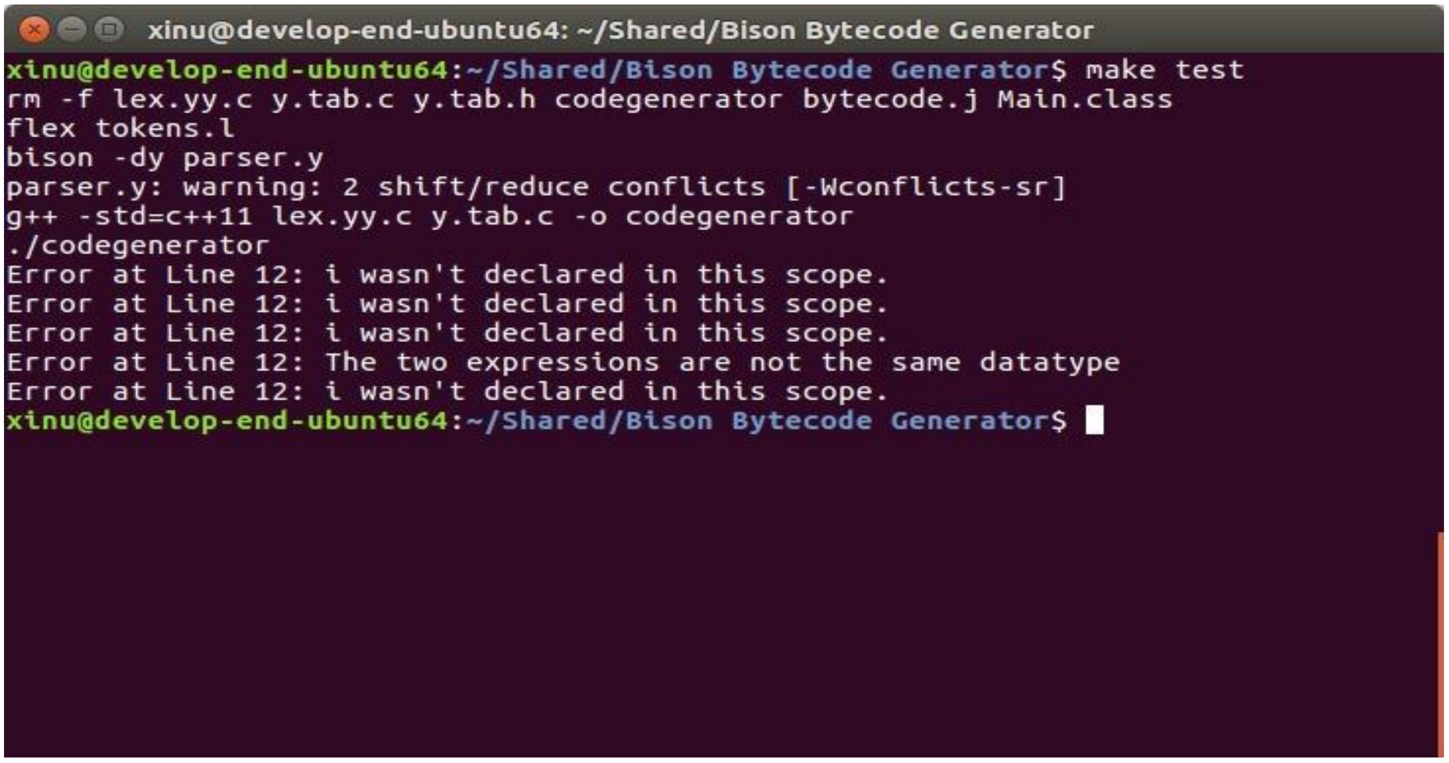
Test case 2 (Faulty program):

Program:



```
input.txt
int x,y,z;
x = 5;
if (x > 2)
{
    y = x;
}
else
{
    x = 0;
    int i;
}
for(i = 0;i<1000;i = i + 1){
    x = x + 10;
}
System.out.println(x);
```

Error handling:



```
xinu@develop-end-ubuntu64: ~/Shared/Bison Bytecode Generator
xinu@develop-end-ubuntu64:~/Shared/Bison Bytecode Generator$ make test
rm -f lex.yy.c y.tab.c y.tab.h codegenerator bytecode.j Main.class
flex tokens.l
bison -dy parser.y
parser.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
g++ -std=c++11 lex.yy.c y.tab.c -o codegenerator
./codegenerator
Error at Line 12: i wasn't declared in this scope.
Error at Line 12: i wasn't declared in this scope.
Error at Line 12: i wasn't declared in this scope.
Error at Line 12: The two expressions are not the same datatype
Error at Line 12: i wasn't declared in this scope.
xinu@develop-end-ubuntu64:~/Shared/Bison Bytecode Generator$
```

Comments about tools:

1- Graph editor:

An online graph editor used to visualize and help debug the graphs used in the project

- NFA
- DFA
- Parsing Tree

(Graph editor)

2- Flex (fast lexical analyzer generator):

Flex (fast lexical analyzer generator) is a tool/program for generating lexical analyzers (scanners or lexers).

Flex was written in C around 1987 by Van Paxson, with the help of many ideas and inspirations from Van Jacobson.

It is usually used together with **Berkeley Yacc parser generator** or **GNU Bison parser generator**.

Supported Languages:

Flex can only generate code using **C** and **C++**.

To use the scanner code generated by flex from other languages a language binding tool such as **SWIG** can be used.

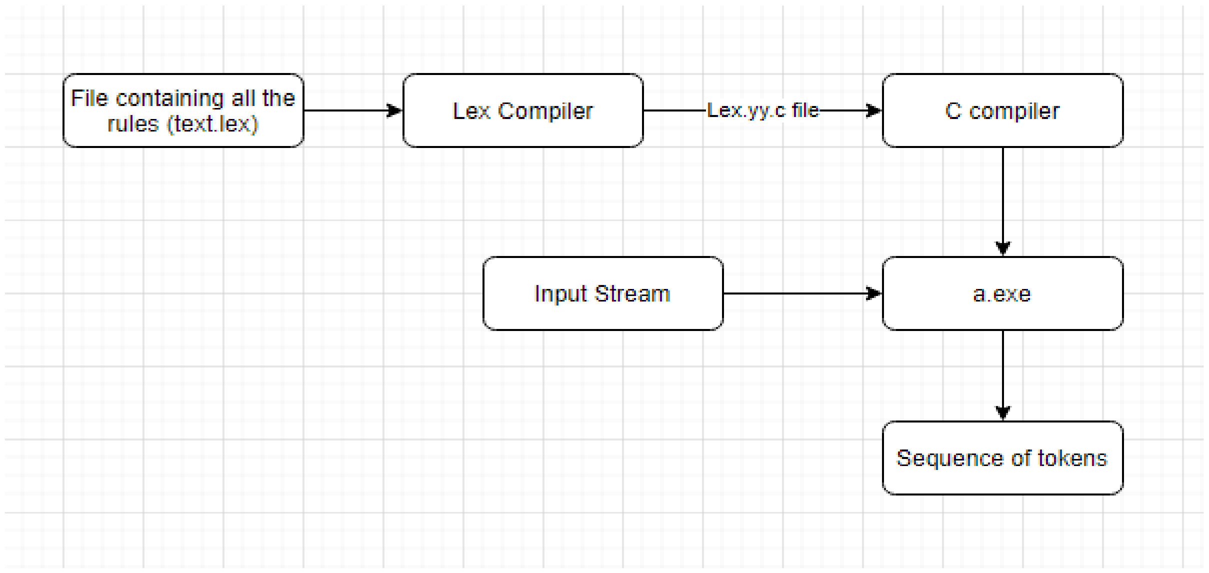


Figure 6: Flex flow

The function $\boldsymbol{yylex}$ () is the main flex function which runs the rules section, and the extension ($\boldsymbol{.l\ or\ .lex}$) is the extension used to save the flex programs.

Program Structure:

In the input file, there are 3 sections:

1. **Definition Section**:

This section is optional, it contains the declarations of variables, regular definitions, manifest constants. In this section, the text is enclosed by %{ %} brackets.

```
%{
      Definitions
%}
```

2. **Rules Section**:

This section is required; it contains a series of rules in the form of patterns and actions. Actions must be on the same line of the corresponding patterns. In this section, the text is enclosed by %% %%

```
%%
      Pattern Action
%%
```

3. **User Code Section**:

This section is optional; it contains C statements and additional functions.

(Fast lexical analyzer generator)

3- Jasmin:

Jasmin is an assembler for the Java virtual machine. It takes ASCII descriptions of Java classes, written in simple assembler-like syntax using the Java virtual machine instruction set (Java bytecode). It converts them into binary java class files, suitable for loading by a Java runtime system.

(Jasmin)

4- Bison:

It is a parser generator that reads a specification of a context free grammar, warns about any parsing ambiguities, and generates a parser which reads sequences of tokens and decides whether the sequence has the correct syntax according to the grammar or not.

Supported Languages:

Flex can only generate code using **C** and **C++**.

Program Structure:

In the input file, there are 3 sections:

1. **Definition Section**:

The C declarations section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of yyparse.

```
%{
      Definitions
%}
```

2. **Bison declarations Section**:

The bison declarations section contains declarations that define terminal and nonterminal symbols.

3. **Rules Section**:

The grammar rules section contains one or more Bison grammar rules, and nothing else., the text is enclosed by %% %%

```
%%
      Pattern Action
%%
```

4. **User Code Section**:

The additional C code is coped verbatim to the end of the parse file. Just as the C declarations section is copied to the beginning. This is the most convenient place to put anything the user want to have in the parser file.

(Bison)

Project internal structure:

1- Every GUI related class is found in the view package
2- Every Lexical analyzer related class is found in the model.lexical 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- Every parser related class is found in the model.parser package
    a. Construction
        i. Parser line processor class
        ii. Parser rules container class
    b. CFG
        i. CFG class (Context free grammar)
    c. Parser
        i. Parser class
        ii. Parser Generator class
        iii. Parsing Tree Node class
4- Bison code generation folder contains the necessary files for flex and bison tools:
    a. Parser.y file
    b. Token.l file

Assumptions:

- The concatenation symbol is **"`"**
- The end marker is the dollar sign **"$"**
- On Eliminating left recursion, we add **"_DASH"** to the new non-terminal
- On Applying left factoring, we add **"_HASH"** to the new non-terminal
- Synchronization token is **"SYNC"**
- All if statements are followed by else statements
- Assignment must be of the same datatype (No casting)

Conclusion:

To summarize, a front-end compiler translates and/or compiles a program written in a suitable source language into the intermediate code representation without changing the meaning of the program through a number of stages.

A front-end compiler constitutes of 3 stages

1- Lexical analyzer
2- Syntax analyzer
3- Semantics analyzer
4- Intermediate code generation

We've learnt various techniques and algorithms that help build a front-end compiler. As well as, some tools such as **Bison**, **Flex**, and **Jasmin.**

Role of each student:

Mohamed Ali Harraz 4608:

Phase 1: DFA generation, and minimization
Phase 2: Parsing table construction, parsing algorithm, and parsing tree generation
Phase 3: Backpatching, if statement, flex-bison linking, boolean expressions

Sherif Rafik 4635:

Phase 1: NFA generation, graph implementation, graph operations, flex report, report
Phase 2: First construction, follow construction, and report
Phase 3: Backpatching, for statement, flex-bison linking, multiple variables declaration, report

Omar Nasr 4730:

Phase 1: Input reading, Tokenization, and GUI
Phase 2: Left recursion elimination, left factoring, and parsing tree generation
Phase 3: Backpatching, symbol table, while statement, variable scopes, system out function

References:

Alfred V. Aho, M. S. (2006). *Compilers Principles, Techniques, and Tools.* U.S: Pearson Education, Inc.

Bison. (n.d.). Retrieved from Disosaur.compilertools: http://dinosaur.compilertools.net/bison/bison_6.html

Elimination of Left Recursion. (n.d.). Retrieved from CSD - Lectures: https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node8.html

Fast lexical analyzer generator. (n.d.). Retrieved from Geeks for geeks: https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/

Graph editor. (n.d.). Retrieved from CS Academy: https://csacademy.com/app/graph_editor/

Jasmin. (n.d.). Retrieved from Jasmin - Sourceforge: http://jasmin.sourceforge.net

Java bytecode - Wikipedia. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

Java SE Specification. (n.d.). Retrieved from Oracle: https://docs.oracle.com/javase/specs/

Left factoring. (n.d.). Retrieved from CSD - Lectures: https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node9.html

Powerset construction - Wikipedia. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Powerset_construction

Thompsons's Construction - Wikipedia. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Thompson%27s_construction