

Syntax Analyzer for “Tiny Language”

<i>Course Code</i> CSE226	<i>Course Name</i> Design of Compilers	
	Semester Spring 2020	Date of Submission 30/05/2020 before 4 pm

#	Student ID
1	17P5026
2	17P6069
3	17P8182
4	17P3061
5	17P8042

Parser Documentation

Table of Contents

Table of Contents

1.0 Introduction	3
2.00 The Scanner Phase	4
2.10 Scientific Background	4
2.11 Description of Scanner Phase	4
2.12 Regular Expressions of two rules	4
2.13 Their equivalent automata	5
2.20 Experimental Results	7
2.21 Examples showing input and output	7
2.22 Examples of errors	8
3.00 The Parser Phase	9
3.10 Scientific Background	9
3.11 Description of Parser Phase	9
3.12 EBNF	9
3.13 Syntax diagrams	10
3.14 Example of left recursive rule	10
3.15 Example of Non-Deterministic rule	10
3.20 Experimental Results	11
2.21 Examples showing input and output	11
2.22 Examples of errors	13
4.00 List of References	14

List of Figures

Figure 1: NFA of 1st example.....	5
Figure 2: DFA of 1st example.....	5
Figure 3: NFA of 2nd example	6
Figure 4: DFA of 2nd example	6
Figure 5: Syntax Diagram (1)	10
Figure 6: Syntax Diagram (2)	10

1. Introduction

The specific purpose of this report is to show a primal description generally for the phases of the compiler, and specifically for the first two phases of it , that we studied through this course, which are “Lexical Analysis” and “Syntax Analysis”, and after this theoretical description we will move to the practical way to justify these theoretical concepts by giving examples from the project that we worked on through the course.

To understand this report properly, we need to have a basic knowledge of programming, software engineering, design automata.

The compiler is the translator that transforms the source program into the targeted one. In 1952 the mother of compilers, as they called her, Grace Hopper has written the first compiler at all, later on, between 1954 to 1957 a complete compiler was developed. In the 1960s, and 1970s there were a lot of theories and algorithms that was developed, or used for the sake of the compilers, as an example of these theories we mention the “Chomsky Hierarchy” which was done by the famous father of the modern linguistics “Noam Chomsky”.

The compiler now a days is consisting of six phases that works sequentially in order to produce the desired code, these phases takes the output of the previous ones as their inputs (except the first one for sure that takes the user input. These seven phases are:

- 1- The Lexical Analysis (Scanner), which reads and analyses the program text by dividing it into understandable tokens.
- 2- The Syntax Analysis (Parser), that takes the output of the Lexical one and orders it to suitable data structure (Syntax Tree) (will talk about these previous phases with more illustration and examples later in the report.)

- 3- The Semantic Analysis, which is a verifier for the syntax one as it checks if the output tree of the parser is meaningful or not and produce the final view of the tree. .It also does type checking, Label checking and Flow control checking.
- 4- Intermediate Code Generation, and this one produces code that can be converted to a machine executable code.
- 5- Code Optimization, which reduces the consumption of resources of the code to be faster in dealing with and in executing. It has two types: 1- Machine Dependent. 2- Machine Independent.
- 6- Target Code Generation, and in this phase the compiler produces the final machine code that the computer can understand and execute.

2. The Scanner Phase

2.10Scientific Background

2.11 Description of Scanner Phase:

The “Lexical Analysis” phase which is the first phase of the compiler is dealing specifically with the input source code characters (strings) and divide them into separate tokens that are meaningful to the compiler, as an example: variable names, numbers, and keywords. After doing this, the tokens will be filtered from what is separating it, as an example: white spaces, and comments.

The grammar rules that is used to define the Lexical syntax is consisting of “regular expressions” which defines the set of possible lexemes of a token. To transfer the regular expressions to efficient programs we use two steps: 1- Nondeterministic Finite Automata (NFA) 2- Deterministic Finite Automata (DFA)

The main purpose of this “Lexical Analysis” process is to facilitate the mission of the next phase “Syntax Analysis”

2.12 Regular Expression of two rules:

Regular Exp for Number:

Digit = [0...9]

number = (Digit)+ [" . " (Digit) +]?

Regular Exp for Identifier

letter = [a-z |A-Z]

digit = [0-9]

Identifier = Letter (Letter | digit)*

2.13 Their equivalent automata (NFA or DFA).:

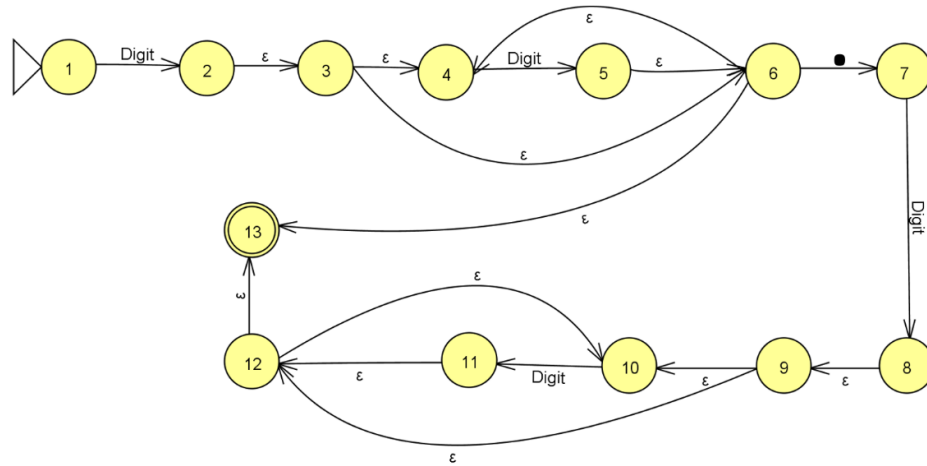


Figure 1: NFA of 1st example

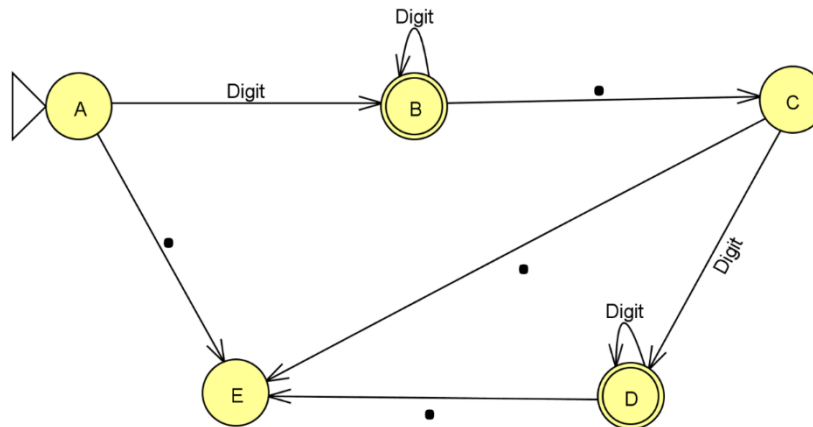


Figure 2: DFA of 1st example

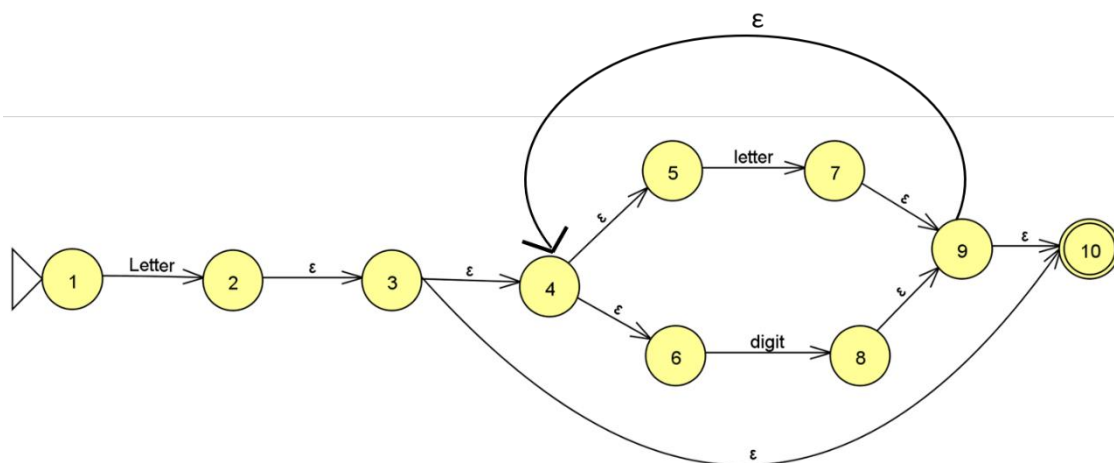


Figure 3: NFA of 2nd example

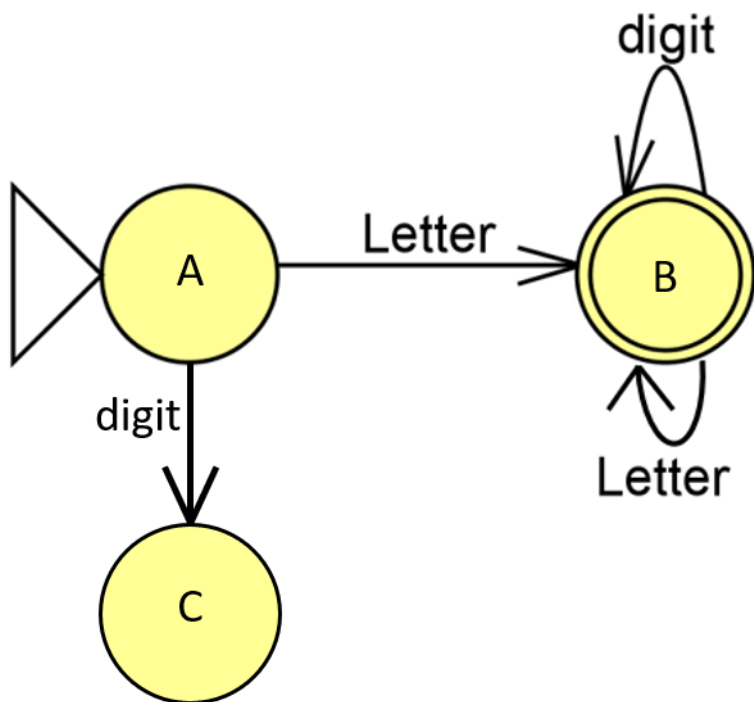


Figure 4: DFA of 2nd example

2.20 Experimental Results

Scanner example 1

CODE	LIST OF TOKENS	LIST OF TOKENS	LIST OF TOKENS	LIST OF TOKENS
<pre>int val, counter; read val; counter:=0; repeat val:=val-1; write "iteration number "; write counter; write "] the value of x = "; write val; write endl; counter:=counter+1; until val = 1 write endl; string s := "number of iterations = "; write s; counter:=counter-1; write counter; /* complicated equation */ float z1 := 3*2*(2+1)/2-5.3; z1:=z1+sum(1/y); if z1 > 5 z1 < counter && z1 = 1 then write z1; elseif z1 < 5 then z1:=5; else z1:=counter; end return 0; }</pre>	<pre>int RESERVED_WORD_int sum Identifier_sum (Separator_ int RESERVED_WORD_int a Identifier_a Separator_ int RESERVED_WORD_int b Identifier_b Separator_ (Operator_ return RESERVED_WORD_return a Identifier_a + Arithmetic_Operator_+ b Identifier_b Separator_) Operator_ int RESERVED_WORD_int main Identifier_main (Separator_) Separator_ int RESERVED_WORD_int val Identifier_val val Identifier_val counter Identifier_counter Separator_ read RESERVED_WORD_read val Identifier_val Separator_)</pre>	<pre>Separator_ counter Identifier_counter + Arithmetic_Operator_+ 1 Number_1 Separator_ until RESERVED_WORD_until val Identifier_val = Operator_ 1 Number_1 write RESERVED_WORD_write end RESERVED_WORD_end Separator_ string RESERVED_WORD_string s Identifier_s := Assign_Operator "number of iterations = " string Separator_ write RESERVED_WORD_write s Identifier_s Separator_ counter Identifier_counter := Assign_Operator counter Identifier_counter + Arithmetic_Operator_+ 1 Number_1 Separator_ write RESERVED_WORD_write counter Identifier_counter Separator_ /* complicated equation */ comment</pre>	<pre>float RESERVED_WORD_float z1 Identifier_z1 := Assign_Operator 3 Number_3 * Arithmetic_Operator_* 2 Number_2 * Arithmetic_Operator_* (Separator_ 2 Number_2 + Arithmetic_Operator_+ 1 Number_1) Separator_ / Arithmetic_Operator_/ 2 Number_2 - Arithmetic_Operator_- 5.3 Number_5.3 Separator_ z1 Identifier_z1 := Assign_Operator z1 Identifier_z1 + Arithmetic_Operator_+ sum Identifier_sum (Separator_ 1 Number_1 Separator_ y Identifier_y) Separator_ Separator_ /* complicated equation */ comment if RESERVED_WORD_if z1 Identifier_z1</pre>	

LIST OF TOKENS	LIST OF TOKENS
<pre>if RESERVED_WORD_if z1 Identifier_z1 > condition_operator> 5 Number_5 Boolean_operator z1 Identifier_z1 < condition_operator< counter Identifier_counter && Boolean_operator&& z1 Identifier_z1 = Operator_ 1 Number_1 then RESERVED_WORD_then write RESERVED_WORD_write z1 Identifier_z1 Separator_ elseif RESERVED_WORD_elseif z1 Identifier_z1 < condition_operator< 5 Number_5 then RESERVED_WORD_then z1 Identifier_z1 := Assign_Operator 5 Number_5 Separator_ else RESERVED_WORD_else z1 Identifier_z1 := Assign_Operator counter Identifier_counter</pre>	<pre>< condition_operator< counter Identifier_counter && Boolean_operator&& z1 Identifier_z1 = Operator_ 1 Number_1 then RESERVED_WORD_then write RESERVED_WORD_write z1 Identifier_z1 Separator_ elseif RESERVED_WORD_elseif z1 Identifier_z1 < condition_operator< 5 Number_5 then RESERVED_WORD_then z1 Identifier_z1 := Assign_Operator 5 Number_5 Separator_ else RESERVED_WORD_else z1 Identifier_z1 := Assign_Operator counter Identifier_counter end RESERVED_WORD_end return RESERVED_WORD_return 0 Number_0 Separator_) Operator_)</pre>

Scanner example 2

Form1

CODE	LIST OF TOKENS	LIST OF TOKENS
<pre>int main() { int x; read x; /*input an integer*/ if x > 0 then /*don't compute if x <= 0 */ int fact := 1; repeat fact := fact * x; x := x - 1; until x = 0 write fact; /*output factorial of x*/ end return 0; }</pre>	<pre>int RESERVED_WORD_int main Identifier_main (Separator_() Separator_) { Operator_(int RESERVED_WORD_int x Identifier_x ; Separator_; read RESERVED_WORD_read x Identifier_x ; Separator_; /*input an integer*/ comment if RESERVED_WORD_if x Identifier_x > condition_operator> 0 Number_0 then RESERVED_WORD_then /*don't compute if x <= 0 */ comment int RESERVED_WORD_int fact Identifier_fact := Assign_Operator 1 Number_1 ; Separator_; repeat RESERVED_WORD_repeat fact Identifier_fact := Assign_Operator fact Identifier_fact * Arithmetic_Operator_* x Identifier_x ; Separator_; x Identifier_x := Assign_Operator x Identifier_x 1 Number_1 ; Separator_; until RESERVED_WORD_until x Identifier_x = Operator_ = 0 Number_0 write RESERVED_WORD_write fact Identifier_fact ; Separator_; /*output factorial of x*/ comment end RESERVED_WORD_end return RESERVED_WORD_return 0 Number_0 ; Separator_;) Operator_)</pre>	<pre>fact Identifier_fact := Assign_Operator 1 Number_1 ; Separator_; repeat RESERVED_WORD_repeat fact Identifier_fact := Assign_Operator fact Identifier_fact * Arithmetic_Operator_* x Identifier_x ; Separator_; x Identifier_x := Assign_Operator x Identifier_x 1 Number_1 ; Separator_; until RESERVED_WORD_until x Identifier_x = Operator_ = 0 Number_0 write RESERVED_WORD_write fact Identifier_fact ; Separator_; /*output factorial of x*/ comment end RESERVED_WORD_end return RESERVED_WORD_return 0 Number_0 ; Separator_;) Operator_)</pre>

SCAN

PARSE

Error 1:

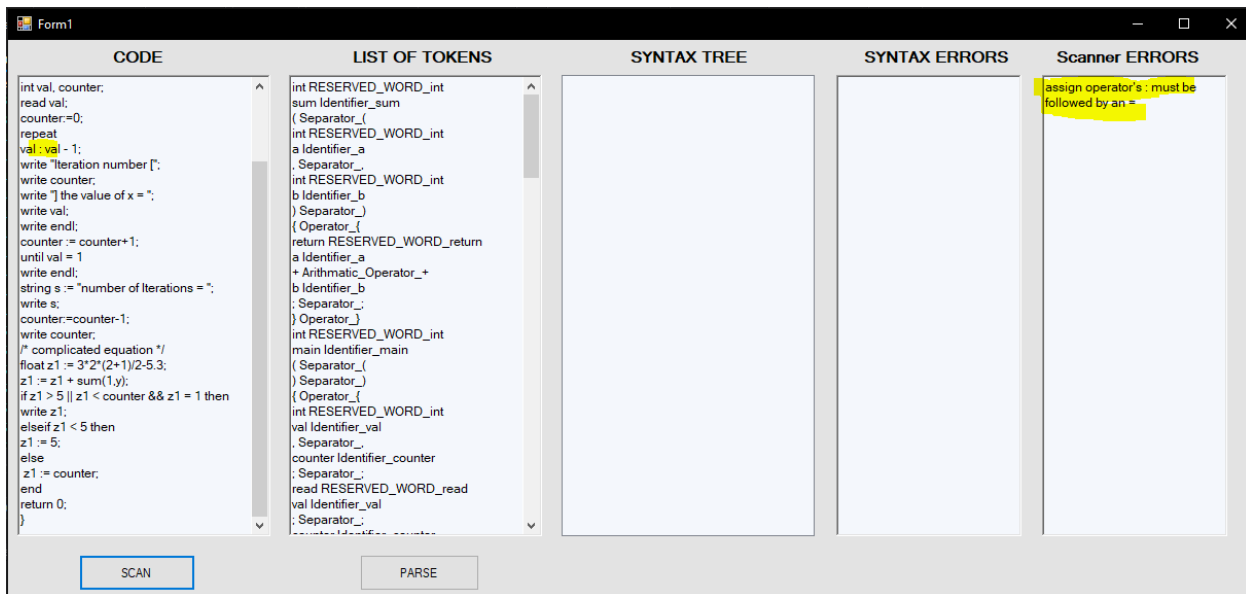
Form1

CODE	LIST OF TOKENS	SYNTAX TREE	SYNTAX ERRORS	Scanner ERRORS
<pre>int sum(int a, int b) { return a + b; } int main() { int 3val, counter; read val; counter:=0; repeat val := val + 1; write "Iteration number ["; write counter; write "] the value of x = "; write val; write endl; counter:= counter+1; until val = 1 write endl; string s := "number of Iterations = "; write s; counter:=counter-1; write counter; /* complicated equation */ float z1 := 3*2*(2+1)/2-5.3; z1 := z1 + sum(1,y); if z1 > 5 z1 < counter && z1 = 1 then write z1; elseif z1 < 5 then z1 := 5; }</pre>	<pre>int RESERVED_WORD_int sum Identifier_sum (Separator_(int RESERVED_WORD_int a Identifier_a ; Separator_ int RESERVED_WORD_int b Identifier_b) Separator_) { Operator_(return RESERVED_WORD_return a Identifier_a + Arithmetic_Operator_+ b Identifier_b ; Separator_) Operator_) int RESERVED_WORD_int main Identifier_main (Separator_() Separator_) { Operator_(int RESERVED_WORD_int ; Separator_ counter Identifier_counter ; Separator_ read RESERVED_WORD_read val Identifier_val ; Separator_ counter Identifier_counter := Assign_Operator 1 Number_1 ; Separator_) Operator_)</pre>			<p>Identifier 3val can't begin with number. Identifier must begin with letter only.</p>

SCAN

PARSE

Error 2



Form1

CODE	LIST OF TOKENS	SYNTAX TREE	SYNTAX ERRORS	Scanner ERRORS
<pre> int val, counter; read val; counter:=0; repeat val:=val - 1; write "Iteration number "; write counter; write " the value of x = "; write val; write endl; counter := counter+1; until val = 1 write endl; string s := "number of iterations = "; write s; counter:=counter-1; write counter; /* complicated equation */ float z1 := 3*2*(2+1)/2-5.3; z1 := z1 + sum(1,y); if z1 > 5 z1 < counter && z1 = 1 then write z1; elseif z1 < 5 then z1 := 5; else z1 := counter; end return 0; </pre>	<pre> int RESERVED_WORD_int sum Identifier_sum (Separator_(int RESERVED_WORD_int a Identifier_a . Separator_(int RESERVED_WORD_int b Identifier_b) Separator_(Operator_(return RESERVED_WORD_return a Identifier_a + Arithmetic_Operator_+ b Identifier_b) Separator_(Operator_(int RESERVED_WORD_int main Identifier_main (Separator_() Separator_(Operator_(int RESERVED_WORD_int val Identifier_val . Separator_(counter Identifier_counter) Separator_(read RESERVED_WORD_read val Identifier_val) Separator_(</pre>			<p>assign operator's : must be followed by an =</p>

SCAN PARSE

3. The Parser Phase

2.1 Scientific Background

- **Description of the parser phase:**

Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as parsing) is to recombine these tokens to recover the intended structure of a program. Not back into a list of characters, but into something that reflects the structure of the text. This “something” is typically a data structure called the syntax tree of the text. It is a tree structure with the leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labelled. In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors. As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. This process is called parser generation. The notation we use for describing the programming language is context-free grammars (EBNF), which is a recursive notation (but not left recursive grammars) for describing sets of strings and imposing a structure on each such string. This is a top down recursive descent parser. Write the EBNF form of the Tiny Grammar.

- **EBNF grammar rules:**

- statement -> if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
- stmt-seq -> statement { ; statement }
- if-stmt -> if (exp) then stmt-seq [else stmt-seq] end

- repeat-stmt -> repeat stmt-seq until exp
- read-stmt -> read identifier
- write-stmt -> write exp
- assign-stmt -> identifier := exp
- exp -> simple-exp [comparison-op simple exp]
- simple-exp -> term {addop term}
- factor -> number | identifier | (exp)
- term -> factor {mulop factor}
- mulop -> * | /
- addop -> + | -
- comparison-op -> < | =

• Syntax Diagrams of two of the Tiny CFG rules:

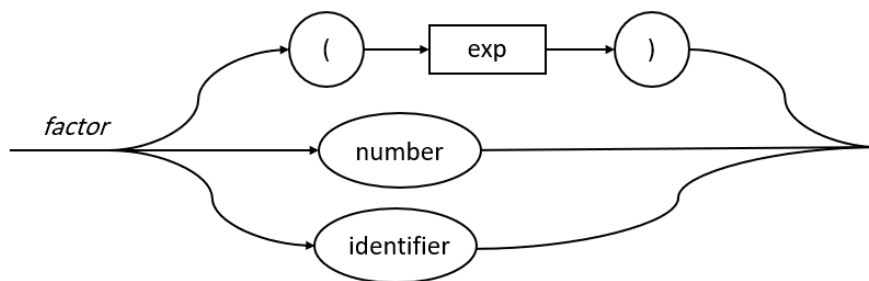


Figure 5: Syntax Diagram (1)

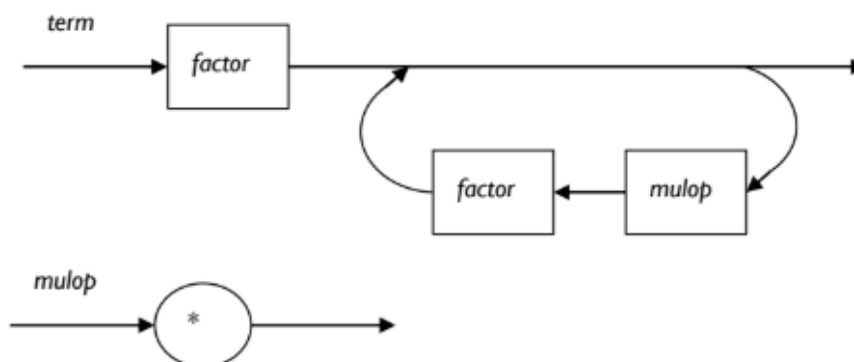


Figure 6: Syntax Diagram (2)

- **Left Recursion Elimination:**

term \rightarrow term mulop factor | factor (left recursive)

SOLUTION \downarrow

Term \rightarrow factor term'

term' $\rightarrow \epsilon$ | mulop factor term'

- **Non Determinism Elimination (Left Factoring):**

if-stmt \rightarrow if exp then stmt-sequence end

|if exp then stmt-sequence else stmt-sequence end

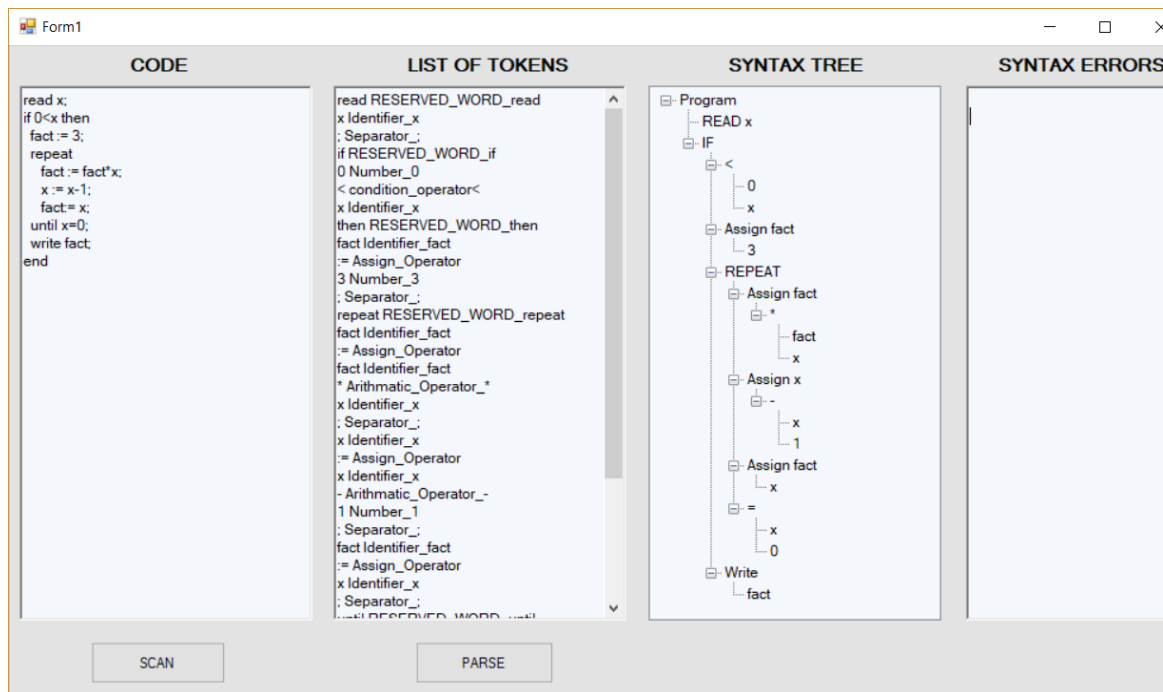
SOLUTION \downarrow

if-stmt \rightarrow if exp then stmt-sequence A

A \rightarrow end | else stmt-sequence end

2.1 Experimental Results

1)Factorial



The screenshot displays a compiler interface with the following components:

- CODE:**

```

read x;
if 0<x then
  fact:= 3;
repeat
  fact:= fact*x;
  x:= x-1;
  fact:= x;
until x=0;
write fact;
end

```
- LIST OF TOKENS:**

```

read RESERVED_WORD_read
x Identifier_x
: Separator_
: RESERVED_WORD_if
0 Number_0
< condition_operator<
x Identifier_x
then RESERVED_WORD_then
fact Identifier_fact
:= Assign_Operator
3 Number_3
: Separator_
repeat RESERVED_WORD_repeat
fact Identifier_fact
:= Assign_Operator
fact Identifier_fact
* Arithmetic_Operator_
x Identifier_x
: Separator_
x Identifier_x
:= Assign_Operator
x Identifier_x
- Arithmetic_Operator_-
1 Number_1
: Separator_
fact Identifier_fact
:= Assign_Operator
x Identifier_x
: Separator_

```
- SYNTAX TREE:**

```

graph TD
    Program --> READ[READ x]
    Program --> IF[IF]
    IF --> LT[<]
    LT --> 0[0]
    IF --> THEN[then]
    THEN --> Assign_fact1[Assign fact]
    Assign_fact1 --> 3[3]
    THEN --> REPEAT[REPEAT]
    REPEAT --> Assign_fact2[Assign fact]
    Assign_fact2 --> MUL[*]
    MUL --> fact1[fact]
    MUL --> x1[x]
    REPEAT --> Assign_x[Assign x]
    Assign_x --> MINUS[-]
    MINUS --> x2[x]
    MINUS --> 1[1]
    REPEAT --> Assign_fact3[Assign fact]
    Assign_fact3 --> EQ[=]
    EQ --> x3[x]
    EQ --> 0[0]
    REPEAT --> Write[Write]
    Write --> fact2[fact]

```
- SYNTAX ERRORS:** (Empty)

Buttons at the bottom: SCAN, PARSE.

2) compute sum of values from 0 to 100

Form1

CODE	LIST OF TOKENS	SYNTAX TREE	SYNTAX
<pre> x:=0; sum:=0; repeat sum:=sum+x; x:=x+1; until x=101; write sum; </pre>	<pre> x Identifier_x := Assign_Operator 0 Number_0 ; Separator_ sum Identifier_sum := Assign_Operator 0 Number_0 ; Separator_ repeat RESERVED_WORD_repeat sum Identifier_sum := Assign_Operator sum Identifier_sum + Arithmetic_Operator_+ x Identifier_x ; Separator_ x Identifier_x := Assign_Operator x Identifier_x + Arithmetic_Operator_+ 1 Number_1 ; Separator_ until RESERVED_WORD_until x Identifier_x = Operator_= 101 Number_101 ; Separator_ write RESERVED_WORD_write sum Identifier_sum ; Separator_ </pre>	<pre> graph TD Program --> Assign_x[Assign x] Assign_x --> 0_0[0] Program --> Assign_sum[Assign sum] Assign_sum --> 0_1[0] Program --> REPEAT REPEAT --> Assign_sum2[Assign sum] Assign_sum2 --> Plus1[+] Plus1 --> sum[sum] Plus1 --> x[x] REPEAT --> Assign_x2[Assign x] Assign_x2 --> Plus2[+] Plus2 --> x2[x] Plus2 --> 1[1] REPEAT --> EQ[=] EQ --> x3[x] EQ --> 101[101] REPEAT --> Write[Write] Write --> sum2[sum] </pre>	

SCAN PARSE

3) swap values of 2 variables

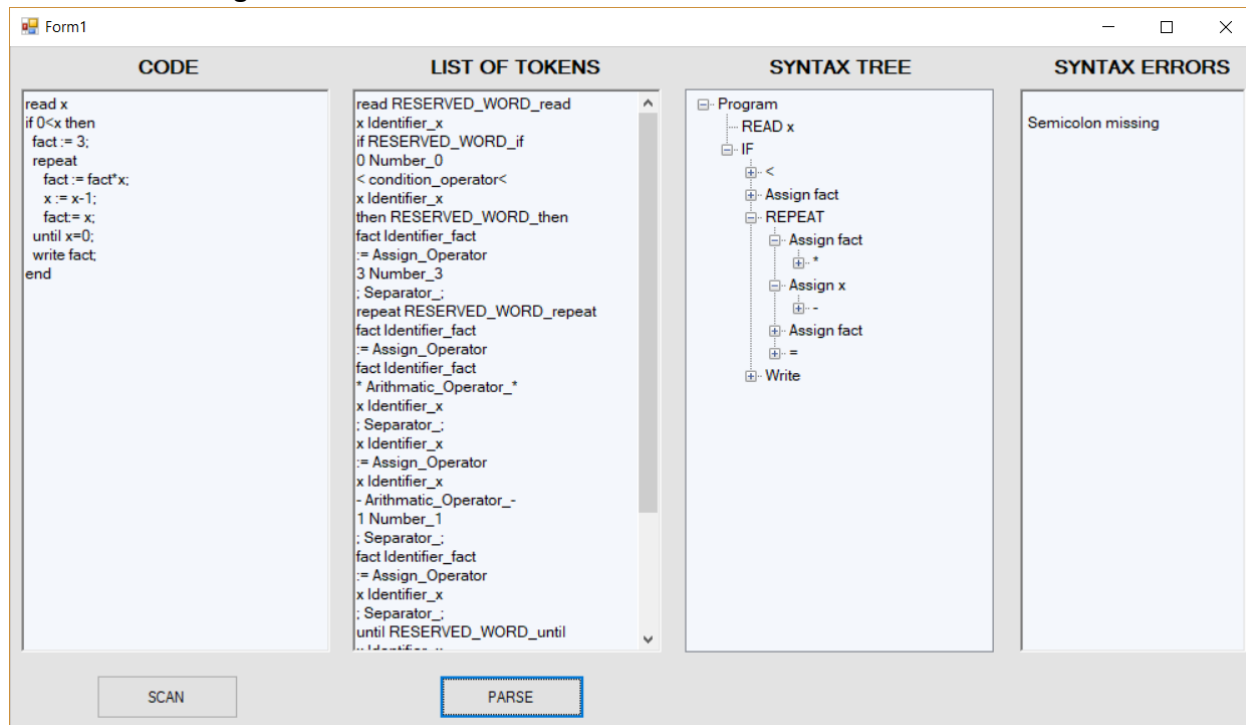
Form1

CODE	LIST OF TOKENS	SYNTAX TREE	SYNTAX ERROR
<pre> read x; read y; temp:=x; x:=y; y:=temp; </pre>	<pre> read RESERVED_WORD_read x Identifier_x ; Separator_ read RESERVED_WORD_read y Identifier_y ; Separator_ temp Identifier_temp := Assign_Operator x Identifier_x ; Separator_ x Identifier_x := Assign_Operator y Identifier_y ; Separator_ y Identifier_y := Assign_Operator temp Identifier_temp ; Separator_ </pre>	<pre> graph TD Program --> READ_x[READ x] Program --> READ_y[READ y] Program --> Assign_temp[Assign temp] Assign_temp --> x[x] Program --> Assign_x[Assign x] Assign_x --> y[y] Program --> Assign_y[Assign y] Assign_y --> temp[temp] </pre>	

SCAN PARSE

Errors

1) Semicolon missing in first line



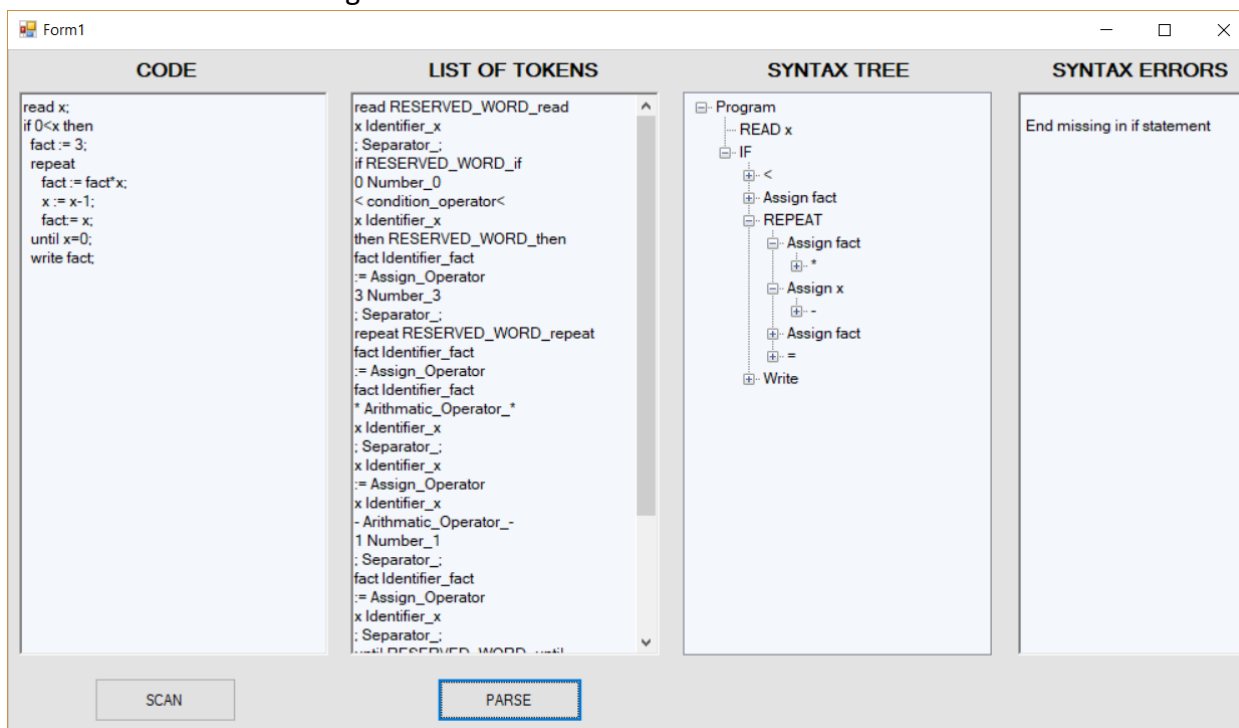
The screenshot shows a compiler interface with four main panels: CODE, LIST OF TOKENS, SYNTAX TREE, and SYNTAX ERRORS. The CODE panel contains the following code:

```
read x
if 0<x then
fact := 3;
repeat
fact := fact*x;
x := x-1;
fact=x;
until x=0;
write fact;
end
```

The LIST OF TOKENS panel shows the tokens generated from the code, including reserved words, identifiers, operators, and separators. The SYNTAX TREE panel shows the parse tree structure, starting with a Program node, followed by a READ x node, an IF node, and a REPEAT node. The SYNTAX ERRORS panel displays the message "Semicolon missing".

Buttons for SCAN and PARSE are located at the bottom of the interface.

2) End of if statement missing in last line:



The screenshot shows a syntax analyzer tool with four panels: CODE, LIST OF TOKENS, SYNTAX TREE, and SYNTAX ERRORS. The CODE panel contains the following code:

```
read x;
if 0<x then
fact := 3;
repeat
fact := fact*x;
x := x-1;
fact = x;
until x=0;
write fact;
```

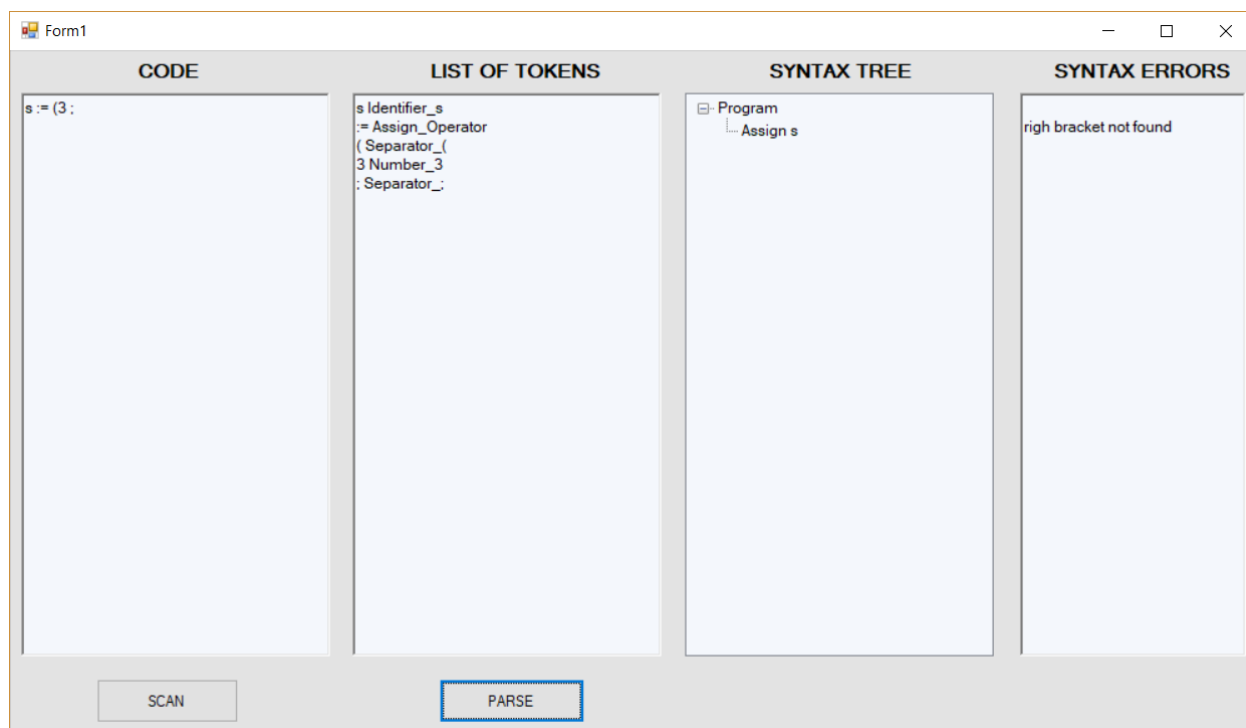
The LIST OF TOKENS panel shows the tokens generated from the code, including reserved words, identifiers, separators, and numbers.

The SYNTAX TREE panel shows the parse tree structure, including the root node 'Program', followed by 'READ x', 'IF', and 'REPEAT' nodes.

The SYNTAX ERRORS panel displays the message: "End missing in if statement".

Buttons for 'SCAN' and 'PARSE' are located at the bottom of the tool.

3) Right bracket not included



The screenshot shows the same syntax analyzer tool with a different input. The CODE panel contains the following code:

```
s := (3;
```

The LIST OF TOKENS panel shows the tokens generated from the code, including identifiers, separators, and numbers.

The SYNTAX TREE panel shows the parse tree structure, including the root node 'Program' and a child node 'Assign s'.

The SYNTAX ERRORS panel displays the message: "right bracket not found".

Buttons for 'SCAN' and 'PARSE' are located at the bottom of the tool.

4. List of References

- 1- Mogensen, Torben. Ægidius. (2010). Basics of Compiler Design. Denmark: Department of Computer Science University of Copenhagen.
- 2- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers: Principles, Techniques, & Tools. Boston: Greg Tobin.
- 3- Loudon, Kenneth. C. (1997). Compiler Construction: Principles and Practice. Boston: PWS Pub.