AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Computer Engineering and Software Systems Program
International Credit Hours Engineering Programs (i.CHEP)

Syntax Analyzer for "Tiny Language"

Course Code CSE226	*Course Name* **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

List of Figures

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).:

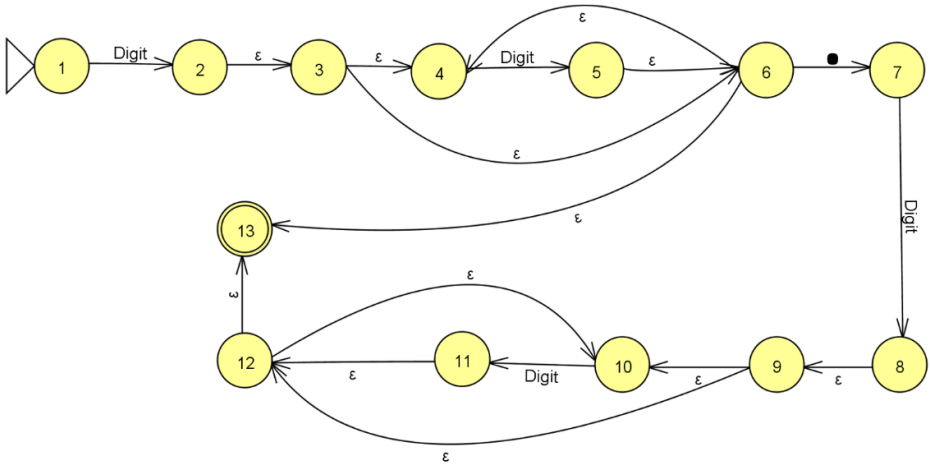


Figure 1: NFA of 1st example

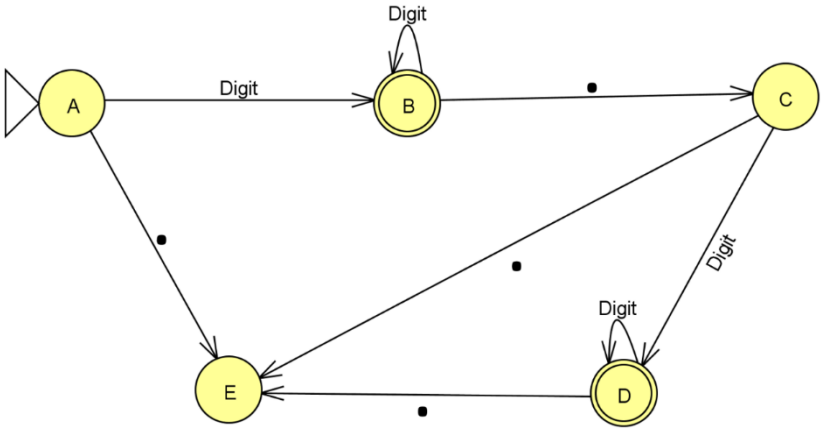


Figure 2: DFA of 1st example

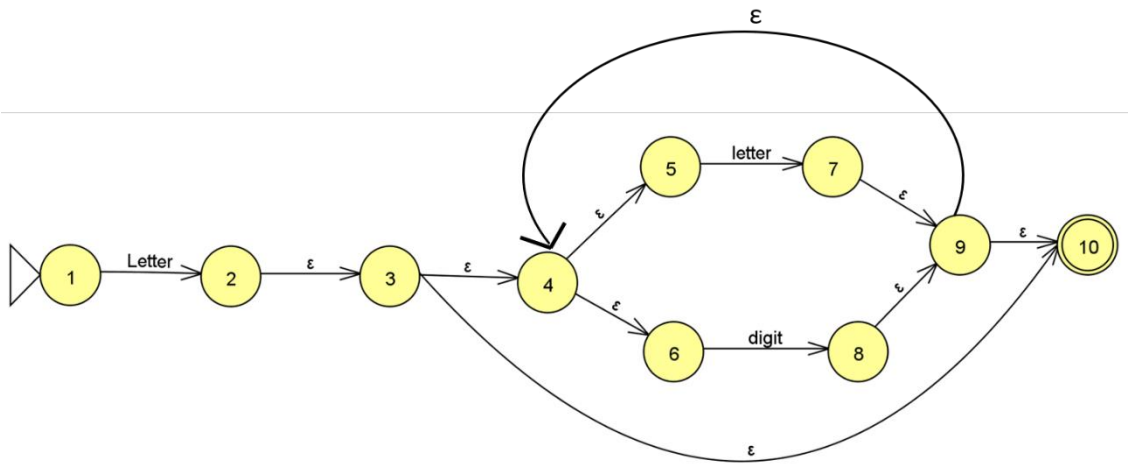


Figure 3: NFA of 2nd example

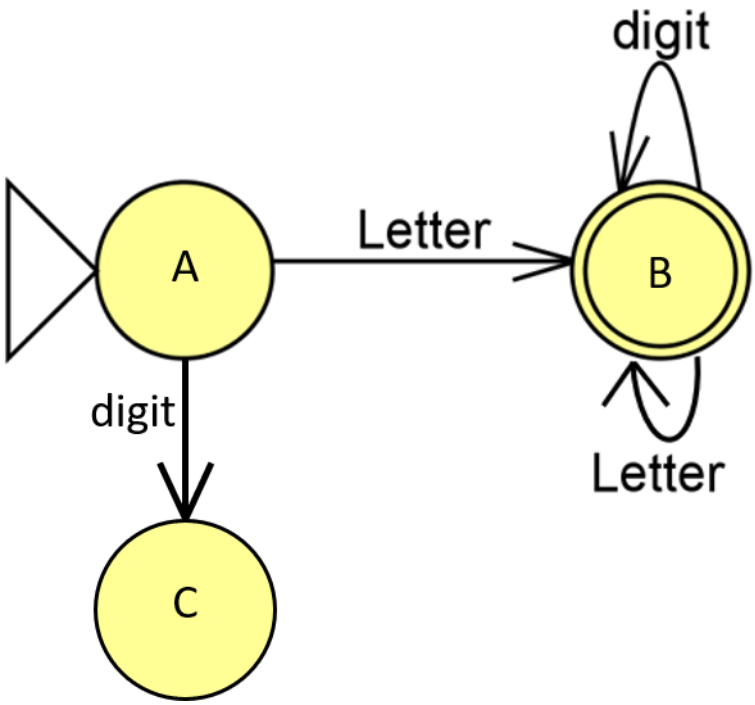
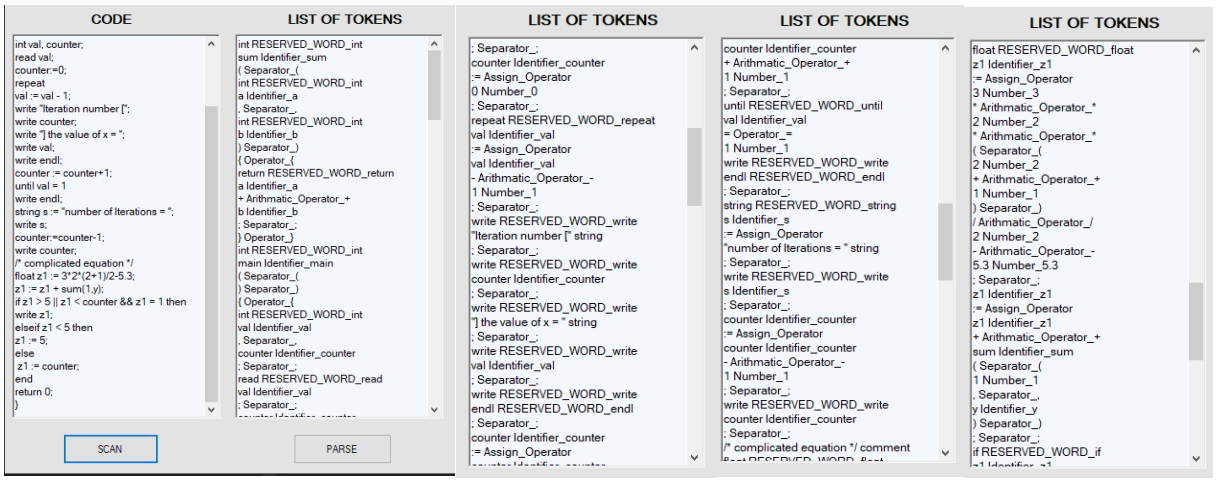
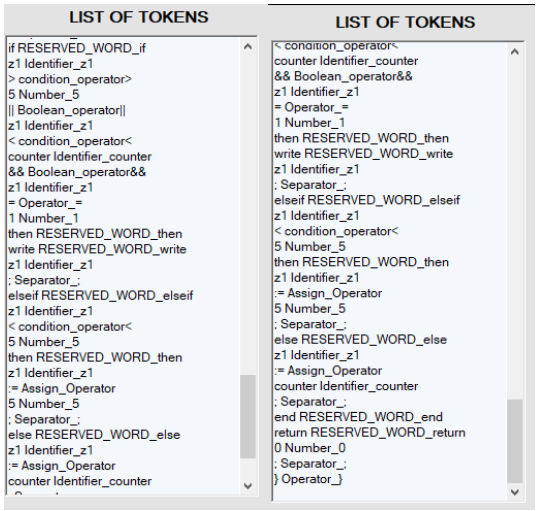


Figure 4: DFA of 2nd example

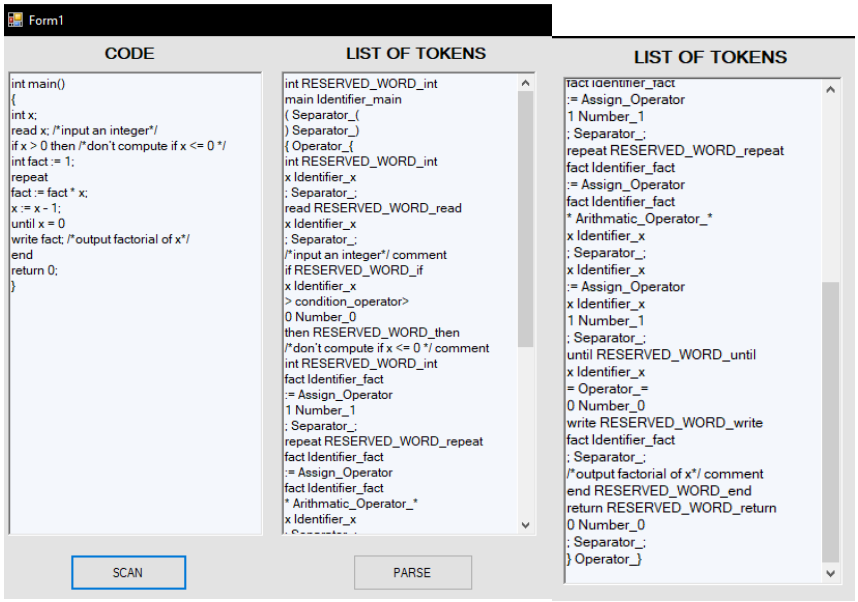
2.20 Experimental Results

Scanner example 1

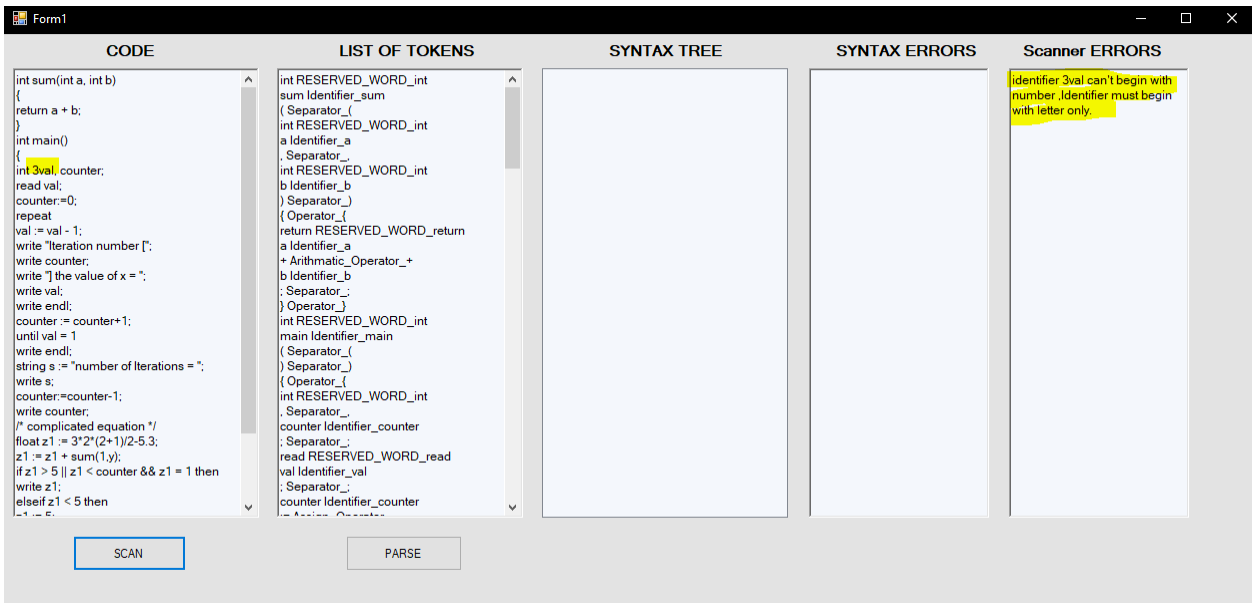




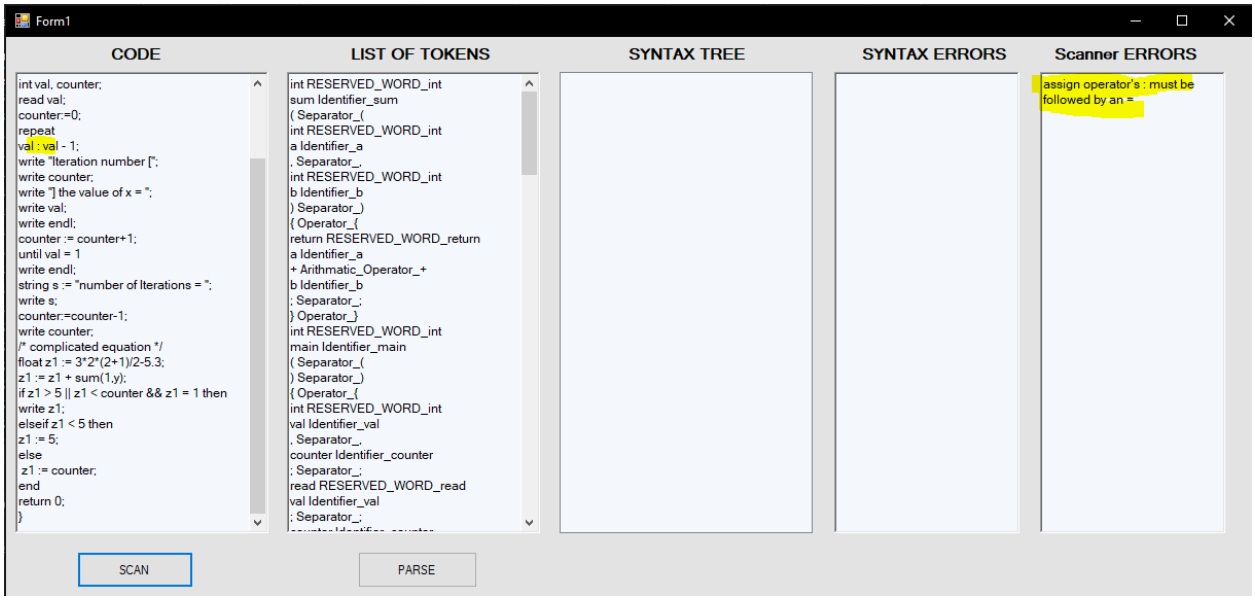
Scanner example 2



Error 1:



Error 2



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 grammer 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:

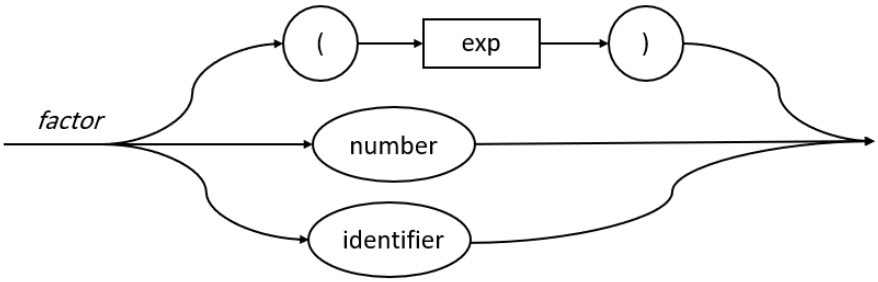


Figure 5: Syntax Diagram (1)

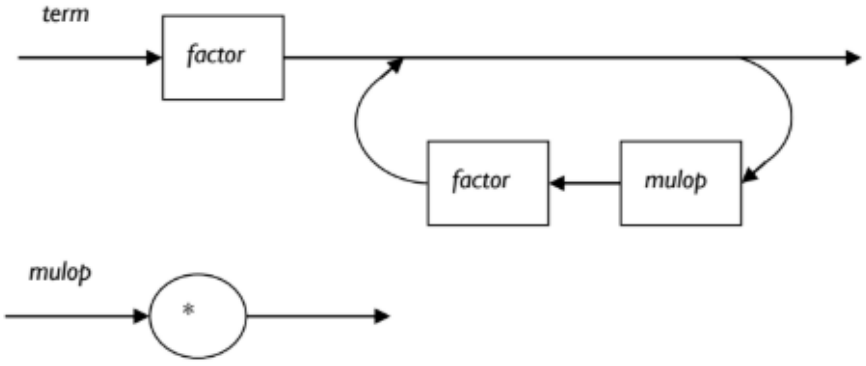


Figure 6: Syntax Diagram (2)

- **Left Recursion Elimination:**

term → term mulop factor | factor (left recursive)

SOLUTION ↓

Term → factor term'

term' → ε | mulop factor term'

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

if-stmt → if exp then stmt-sequence end

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

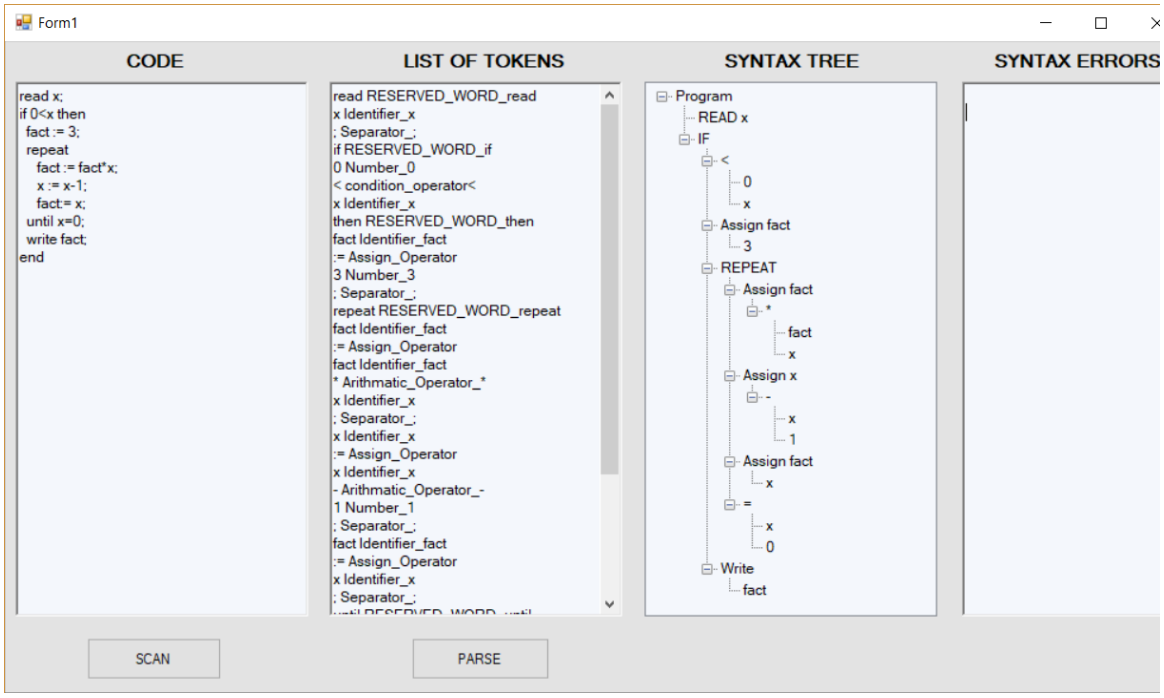
SOLUTION ↓

if-stmt → if exp then stmt-sequence A

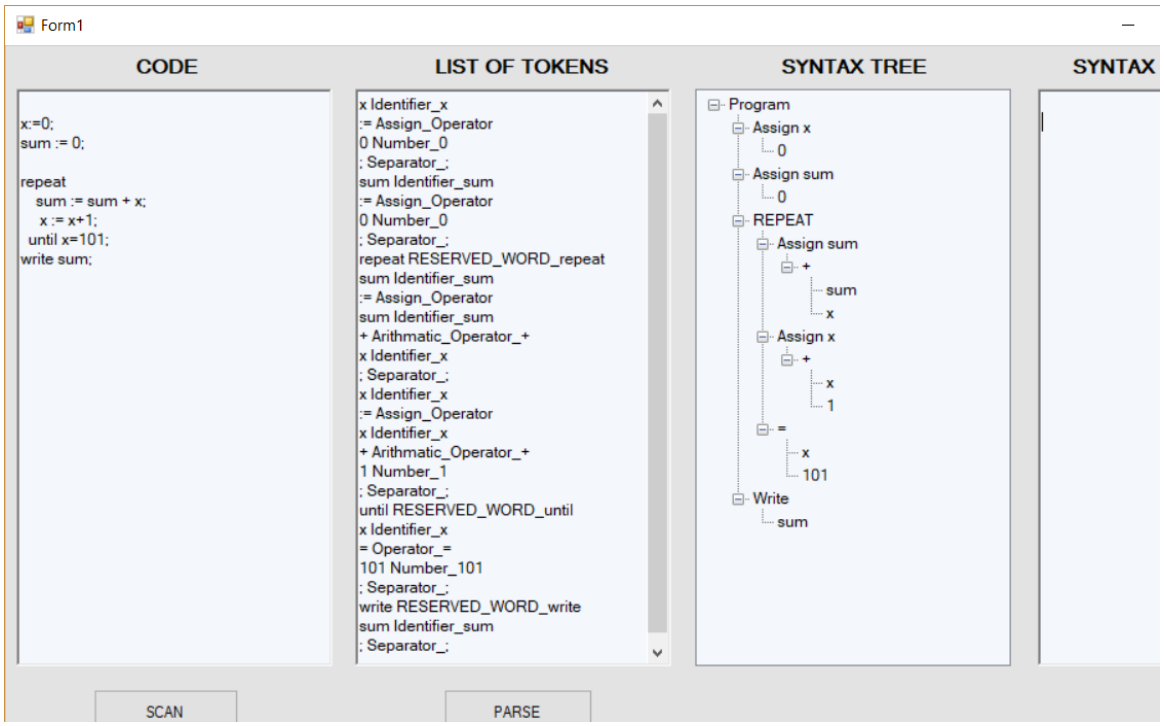
A → end | else stmt-sequence end

2.1 Experimental Results

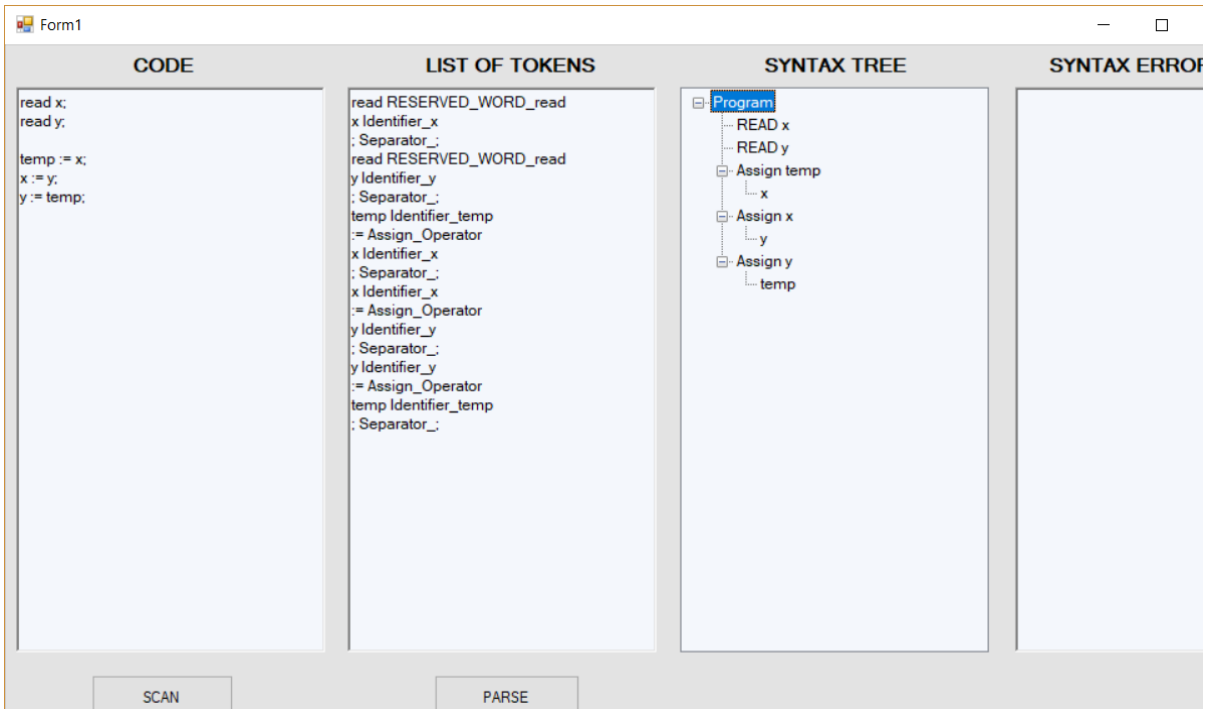
1)Factorial



2) compute sum of values from 0 to 100

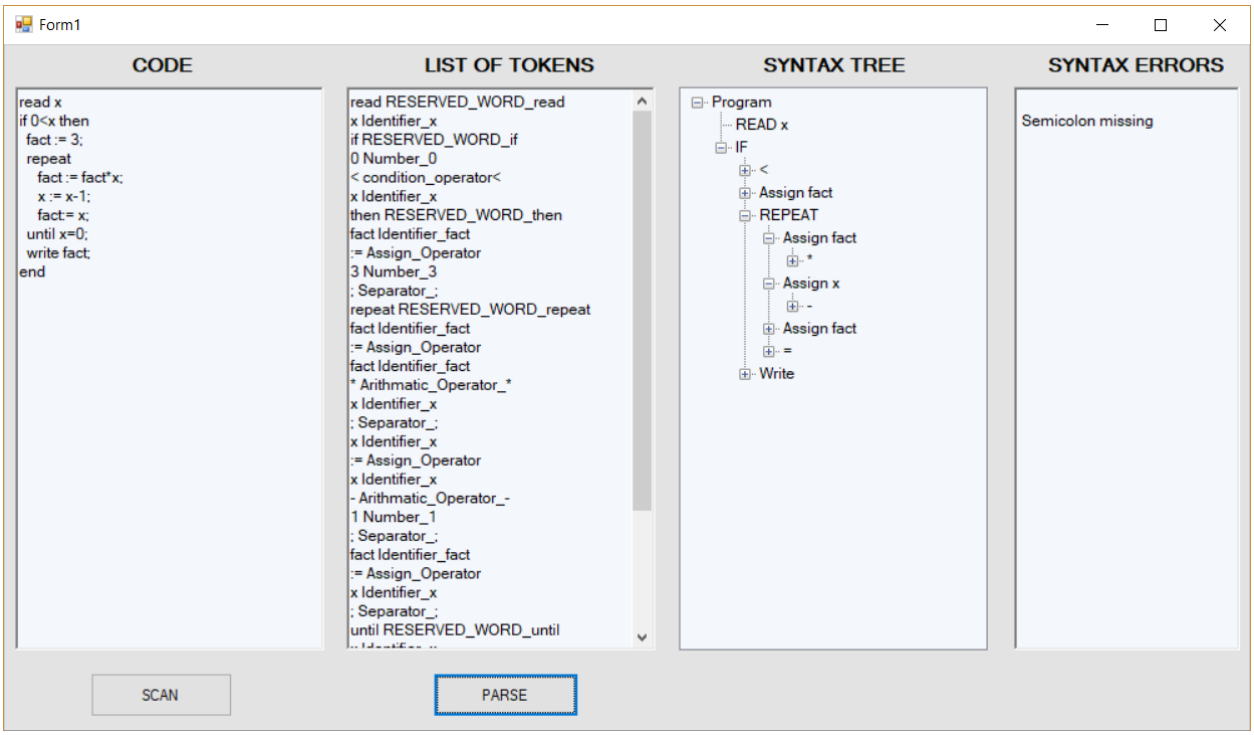


3) swap values of 2 variables

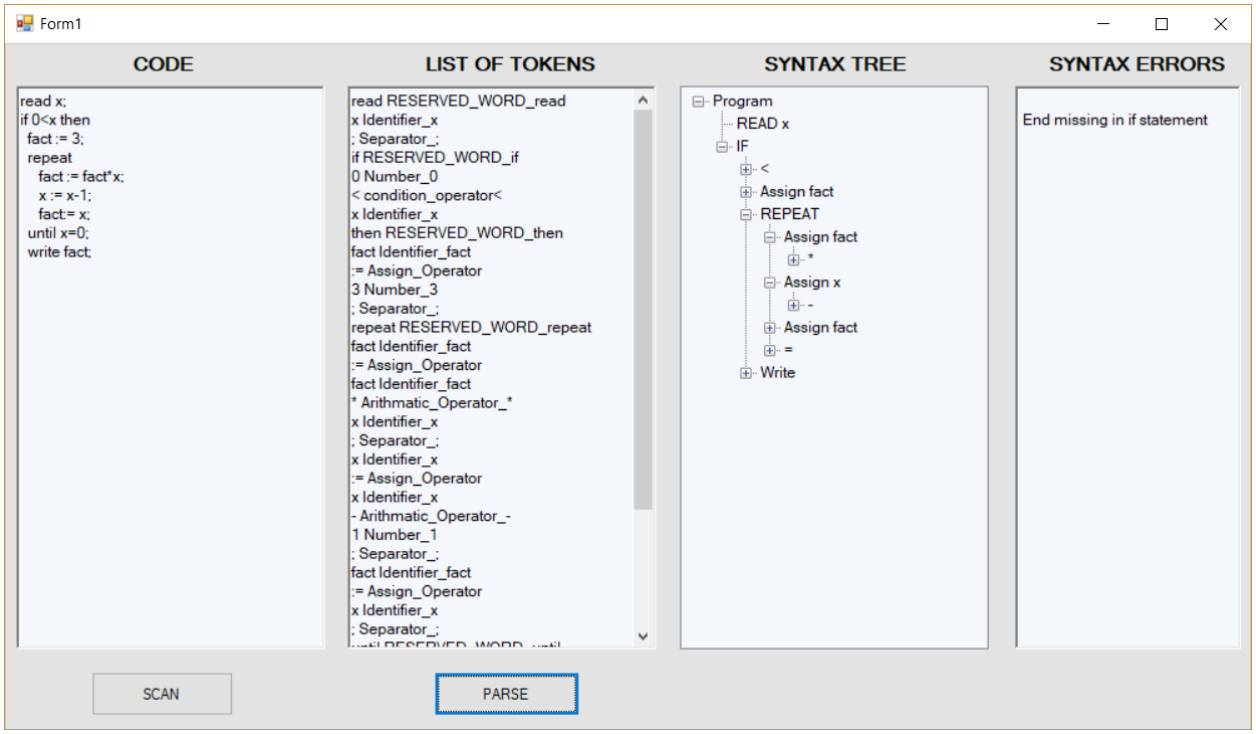


Errors

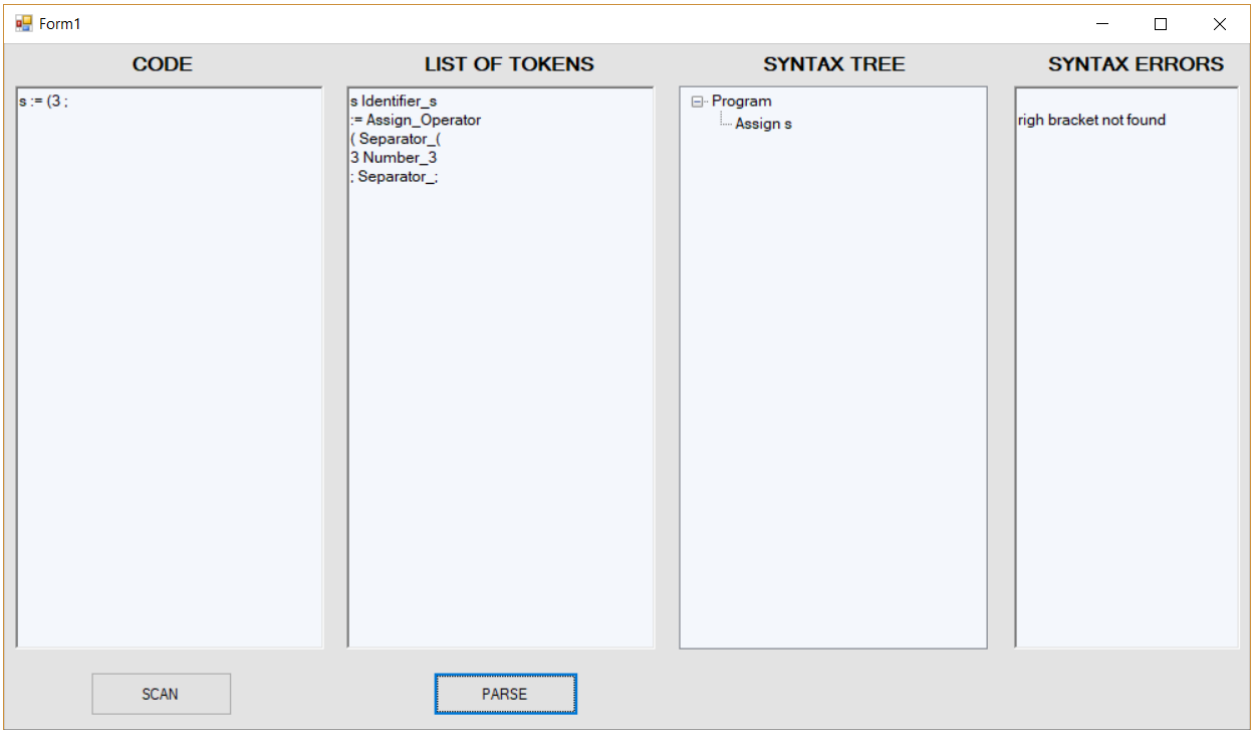
1) Semicolon missing in first line



2) End of if statement missing in last line:



3) Right bracket not included



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- Louden, Kenneth. C. (1997). Compiler Construction: Principles and Practice. Boston: PWS Pub.