

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 | 17P8113 |
| 2 | 17P1048 |
| 3 | 17P6024 |
| 4 | 17P8237 |
| 5 | 15P3041 |

List all students IDs.
DO NOT WRITE STUDENTS NAMES

TABLE OF CONTENT

| | |
|--|-----------|
| TABLE OF CONTENT | 2 |
| LIST OF FIGURES | 3 |
| 1. INTRODUCTION | 4 |
| 2. SCANNER PHASE | 5 |
| 2.1. Scientific Background | 5 |
| Regular expressions with equivalent automata | 7 |
| 2.2. Experimental Results | 8 |
| Sample code I | 8 |
| Sample code II | 9 |
| Sample code with lexical errors | 10 |
| 3. PARSER PHASE | 11 |
| 3.1. Introduction | 11 |
| 3.2. EBNF of The Tiny Grammar | 11 |
| 3.3. Syntax Diagrams | 12 |
| 3.4. Ambiguity | 13 |
| Left recursion rule | 13 |
| Non-deterministic rule | 13 |
| 3.5. Experimental Results | 14 |
| Sample code I | 14 |
| Sample code II | 15 |
| Sample code III | 16 |
| Sample code with syntax errors | 17 |
| REFERENCES | 18 |

LIST OF FIGURES

| | |
|---|-----------|
| Figure 1 Lexical Analysis | 5 |
| Figure 2 Number Regular Expression Automata | 7 |
| Figure 3 String Regular Expression Automata | 7 |
| Figure 4 Comment Regular Expression Automata | 7 |
| Figure 5 Scanner Sample Code I | 8 |
| Figure 6 Scanner Sample Code II | 9 |
| Figure 7 Scanner Sample Code With Errors | 10 |
| Figure 8 stmt-sequence Syntax Diagram | 12 |
| Figure 9 statement Syntax Diagram | 12 |
| Figure 10 if-stmt Syntax Diagram | 12 |
| Figure 11 Parser Sample Code I | 14 |
| Figure 12 Parser Sample Code II | 15 |
| Figure 13 Parser Sample Code III | 16 |
| Figure 14 Parser Sample Code With Errors | 17 |

1. INTRODUCTION

In this report we will go through two of the main compilers phases.

But first we need to define what is a compiler in computer programming. A compiler typically is a program which translates a code of a programming language (known to be the target language) while being written in a language (known to be the source). The term compiler is basically used for creating an executable program for programs that seek to translate source codes written in a high level programming language to a lower level language (e.g. assembly language, object code, or machine code). Although it is some time get used to convert a code in high language to another. [1] (Louden, 1977)

As I've said, in this report we shall discuss two important phases for a compiler: The lexical analysis and the syntax analysis. Also known as scanner and parser. For each we are going discuss a brief scientific background alongside with some examples of each phase steps.

Later on, we will introduce an implementation of the two phases using C# as a source language. At each phase, sample codes will be run for test illustrating their results. Also some errors to be introduced and handled.

That was a quick introduction to what this report is about.

2. SCANNER PHASE

The scanning phase is the first phase of a compiler, also known as the Lexical Analysis phase. It's mainly concerned with converting the high level input code of a program into a sequence of Tokens by a process that is referred to as Tokenization. In a brief description, we will discuss this phase in more details.

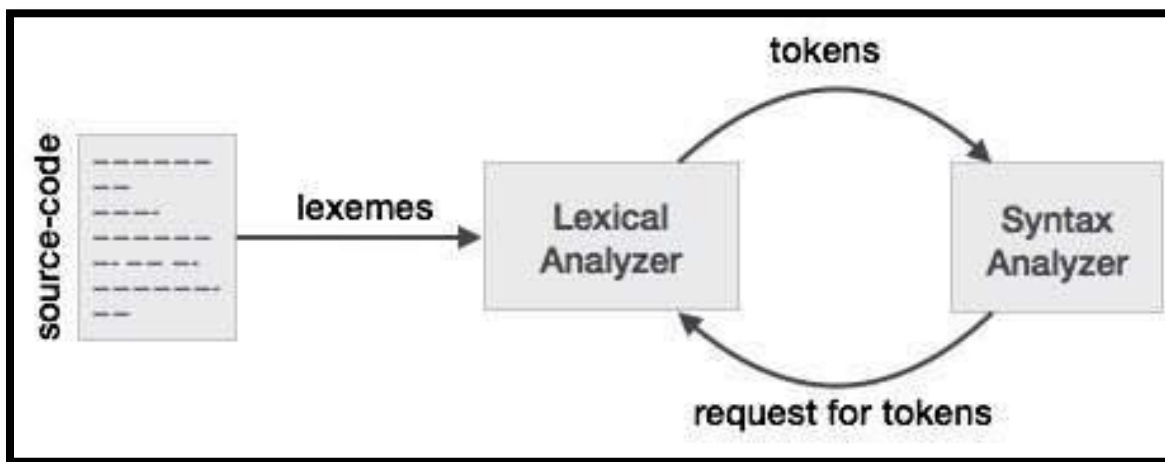


Figure 1 Lexical Analysis

2.1. Scientific Background

We start our discussion by defining the input single unit to the scanner which is Lexeme. Typically, A Lexeme is a sequence of characters in the source code that is matching a pattern to form a corresponding token and is identified by the lexical analyzer as to be an instance of that token. A lexical Token is a string with an assigned and identified meaning. The structure of a token is a pair consisting of a token name usually represented by an enumeration type and an optional token value. Common tokens are identifier, reserved word, separator, operator, literal, and comment. [2] (Introduction of Lexical Analysis, 2020)

The specification of the target programming language includes a set of rules known as the lexical grammar, that defines the lexical syntax. The lexical syntax is usually expressed using regular language, with the grammar rules written in a form of regular expressions that define the set of possible character sequences of a token lexemes are allowed to take in order to be valid. What a scanner does is recognizing strings, and for each kind of string found the lexical program (Scanner) generates a token.

There are two important lexical categories to consider: white space and comments. These are also defined in the lexical grammar, thus they have regular expressions and are processed by the scanner, but in most cases they are to be discarded and don't produce any tokens.

Tokenization simply is the process of generating or producing tokens by the scanner by demarcating and classifying some sections of the input string characters. Before passing the resulting tokens to the Symantic analyser (Parser), tokens may be passed on to some other form of processing.

We identify tokens based on the specific rules of the scanner. Some ways used to identify tokens include: regular expressions that express lexical grammar, specific sequences of characters, specific separating characters called delimiters (like the space and line feed), and explicit definition by a dictionary. There are special characters that include punctuation characters.

A lexical analyzer (scanner) generally does nothing with the different combinations of tokens, such task is left for a parser. i.e, the scanner recognizes parentheses as tokens, but it's not its mission to ensure that each open paranthese is matched with a closing one.

The representation used to feed the scanned tokens to the parser is typically an enumerated list of number representations. If the scanner finds an invalid token (a lexemt that doesn't match any token), it reports an error – usually in the form of Error token – as we will see in some examples. [3] (Terry, 2004)

The scanner, is usually based on a finite-state machine (FSM) which encodes within it the information needed to identify the possible sequences of characters that can be contained within a token. Here's how it works: In most cases, the first appearing non-whitespace character can be used to deduce or predict the kind of token. Then a subsequent input characters are scanned until we reach a character that is not in the set of acceptable ones for a token. At this time we can say we have a lexeme that matches a defined token. This method is called maximal munch, or longest match rule.

A lexeme, however, is only a string of characters known to be of a certain kind. In order to produce a token with typical construction, the lexical analyzer needs a second stage, the **Evaluator**, which runs over the lexeme characters to give it a corresponding real value rather than being just a sequence of string characters. The lexeme's type combined with its value is what properly constitutes a token, which later can be passed to a parser. On the other hand, some tokens do not have real values such as parentheses. And so an evaluation for these will return nothing: only the type is needed. Some times we defer the evaluating process to the symantic analysis phase. But in our case here it's combined with the scanner. The evaluator should produce a value that matches the corresponding token type. i.e, for an integer number it produces an integer value. While in some cases it only makes some modivications rather than call it evaluating. For a simple quoted string literal, the evaluator only needs to remove the quotes. Further details on this to be introduced when we present the experimental results. [4] (Lexical analysis, 2020)

Regular expressions with equivalent automata

(1) Number: $\text{digit} = [0-9]$ $\text{nat} = (\text{digit})^+$ $\text{Number} = \text{nat}(\text{"."nat})?$

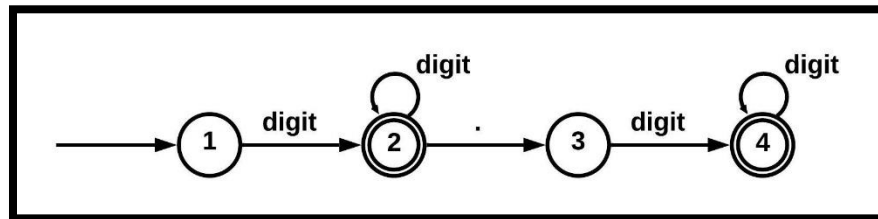


Figure 2 Number Regular Expression Automata

(2) String: letter = [A-Z | a-z] digit = [0-9] String = “ “ ” . * “ ”

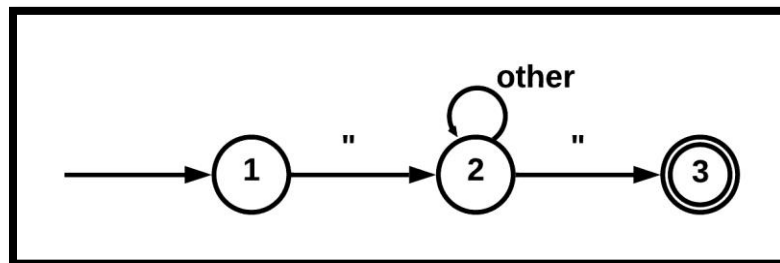


Figure 3 String Regular Expression Automata

(3) Comment: “/*” . * “*/”

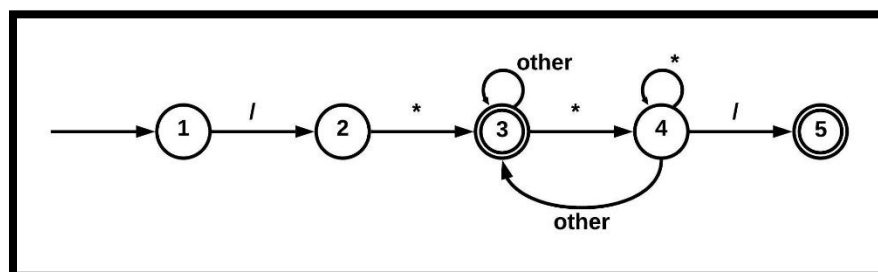


Figure 4 Comment Regular Expression Automata

2.2. Experimental Results

We will scan 2 code samples provided in the project description to test the scanner, showing snapshots of the resultant lexeme/token generated table.

Sample code I

TINY LANGUAGE SCANNER

TYPE CODE or just relax and UPLOAD FILE

```

/*Sample program includes all rules*/
int sum(int a, int b)
{
    return a + b;
}
int main()
{
    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 */

```

Scan Clear

Ver 1.3.0504 All rights reserved

| Lexeme | Token |
|--------|--------------------|
| int | DATATYPE_INT |
| sum | IDENTIFIER |
| (| OPEN_PARENTHESSES |
| int | DATATYPE_INT |
| a | IDENTIFIER |
| , | COMMA |
| int | DATATYPE_INT |
| b | IDENTIFIER |
|) | CLOSE_PARENTHESSES |
| { | OPEN_BRACES |
| return | RESWORD_RETURN |
| a | IDENTIFIER |
| + | PLUS |
| b | IDENTIFIER |
| ; | SIMICOLOCN |
| } | CLOSE_BRACES |

| Error | Error Type |
|-------|------------|
|-------|------------|

Figure 5 Scanner Sample Code I

Sample code II

```
/* Sample program in Tiny language - computes factorial*/
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;
}
```

TINY LANGUAGE SCANNER

TYPE CODE

or just relax and

UPLOAD FILE

```
/* Sample program in Tiny language - computes factorial*/
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;
}
```

Scan

Clear

Ver 1.3.0504 All rights reserved

| Lexeme | Token |
|--------|-------------------|
| int | DATATYPE_INT |
| main | RESWORD_MAIN |
| (| OPEN_PARENTHESES |
|) | CLOSE_PARENTHESES |
| { | OPEN_BRACES |
| int | DATATYPE_INT |
| x | IDENTIFIER |
| ; | SIMICOLOCN |
| read | RESWORD_READ |
| x | IDENTIFIER |
| ; | SIMICOLOCN |
| if | RESWORD_IF |
| x | IDENTIFIER |
| > | GREATER |
| 0 | NUMBER |
| then | RESWORD_THEN |

| Error | Error Type |
|-------|------------|
|-------|------------|

Figure 6 Scanner Sample Code II

Sample code with lexical errors

We will introduce some errors in the code and detect them with the scanner showing the error lexeme and also the error type. Note the errors in the errors Table.

Errors introduced (Noted in the code):

- Unrecognized character (doesn't match any language defined token)
- Floating point error
- Runaway string
- Runaway comment

```
/* Sample code with errors */
int main() {
float x;
x := 15.    /* Floating point error */
x # 10;      /* = unrecognized */
String s;
/* Runaway string error */
s := "ABCD
/* This is a runaway comment *
```

TINY LANGUAGE SCANNER

TYPE CODE or just relax and UPLOAD FILE

```
/* Sample code with errors */

int main()
{
float x;
x := 15.    /* Floating point error */
x # 10;      /* = unrecognized */

String s;
/* Runaway string error */
s := "ABCD
/* This is a runaway comment *
```

Scan Clear

Ver 1.3.0504 All rights reserved

| Lexeme | Token |
|--------|---------------------|
| int | DATATYPE_INT |
| main | RESWORD_MAIN |
| (| OPEN_PARENTHESSES |
|) | CLOSE_PARENTHESSES |
| { | OPEN_BRACES |
| float | DATATYPE_FLOAT |
| x | IDENTIFIER |
| ; | SIMICOLOCN |
| x | IDENTIFIER |
| := | ASSIGNMENT_OPERATOR |
| x | IDENTIFIER |
| 10 | NUMBER |
| ; | SIMICOLOCN |
| String | IDENTIFIER |
| s | IDENTIFIER |
| ; | SIMICOLOCN |

| Error | Error Type |
|--------------------------------|----------------------|
| 15. | FLOATING_POINT_ERROR |
| # | UN_RECOGNIZED_CHAR |
| "ABCD | RUNAWAY_STRING |
| /* This is a runaway comment * | RUNAWAY_COMMENT |

Figure 7 Scanner Sample Code With Errors

3. PARSER PHASE

3.1. Introduction

After splitting the user input to tokens the role of parsing phase start that it must recombine the tokens but not in the last order. It combines it to show the structure of text.

Structure of text is the syntax tree concluded from the language I want to compile. The leaves of this tree are the tokens defined before read from left to right the same as input text. The role of the syntax tree is how these leaves are combined to form the structure of the tree and the label of the interior nodes. In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors. [5] (Appel, 1998)

3.2. EBNF of The Tiny Grammar

In this part, the EBNF of tiny ode which is a way to present language grammar with more interest in repetition and choices.

The tiny grammar EBNF:

| | |
|------------------------|---|
| <i>program</i> → | <i>stmt-sequence</i> |
| <i>stmt-sequence</i> → | <i>statement</i> { ; <i>statement</i> } |
| <i>statement</i> → | <i>if-stmt</i> <i>repeat-stmt</i> <i>assign-stmt</i> <i>read-stmt</i> <i>write-stmt</i> |
| <i>if-stmt</i> → | if <i>exp</i> then <i>stmt-sequence</i> [else <i>stmt-sequence</i>] end |
| <i>repeat-stmt</i> → | repeat <i>stmt-sequence</i> until <i>exp</i> |
| <i>assign-stmt</i> → | identifier := <i>exp</i> |
| <i>read-stmt</i> → | read <i>identifier</i> |
| <i>write-stmt</i> → | write <i>exp</i> |
| <i>exp</i> → | <i>simple-exp</i> { <i>comparison-op</i> <i>simple-exp</i> } |
| <i>comparison-op</i> → | > < = |
| <i>simple-exp</i> → | <i>term</i> { <i>addop</i> <i>term</i> } |
| <i>addop</i> → | + / |
| <i>term</i> → | <i>factor</i> { <i>mulop</i> <i>factor</i> } |
| <i>mulop</i> → | * / |
| <i>factor</i> → | (<i>exp</i>) number <i>identifier</i> |

3.3. Syntax Diagrams

stmt-sequence \rightarrow *statement* { ; *statement* }

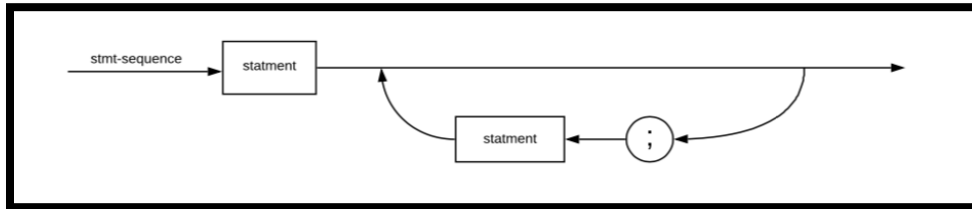


Figure 8 *stmt-sequence* Syntax Diagram

statement \rightarrow *if-stmt* / *repeat-stmt* / *assig-stmt* / *read-stmt* / *write-stmt*

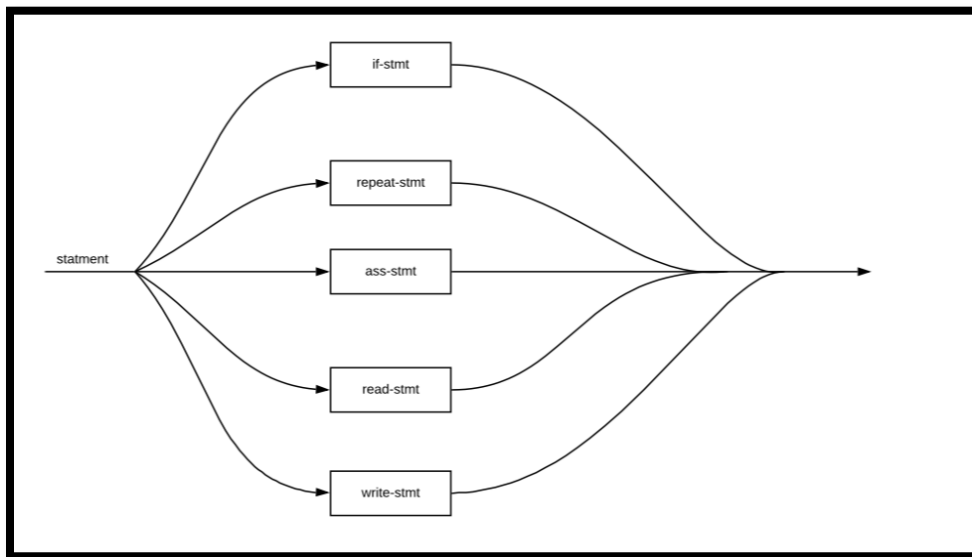


Figure 9 *statement* Syntax Diagram

if-stmt \rightarrow **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**

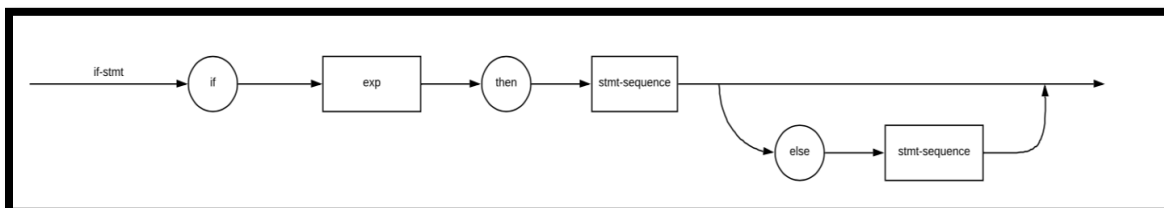


Figure 10 *if-stmt* Syntax Diagram

3.4. Ambiguity

While designing the syntax analysis of any grammar we need to minimize the ambiguity of rules to make it clear. Tiny parser will be top down which need to ignore left recursion and non-determinism.

Left recursion rule

Left recursion is a way to represent repetition availability in grammar it uses a defined part with the part again and terminals can be used.

Ex) $A \rightarrow A \text{ alpha} / B$ So, this statement can be: $\{B \text{ alpha}, B \text{ alpha alpha}, B \text{ alpha alpha alpha}, \dots\}$

This statement can be resolved to: $A \rightarrow B A'$ $A' \rightarrow \text{alpha} A' \mid \epsilon$

A top-down parser cannot handle left recursive productions, the following examples explains why.

(1) $S \rightarrow a$ (2) $S \rightarrow S a$

The parser will not know if there is **a** after **S** in (2). So rule must be resolved to handle this problem.

Left recursion example in Tiny: $\text{Stmt-sequence} \rightarrow \text{stmt-serquence} ; \text{statement} / \text{statement}$

Resolution: $\text{stmt-sequence} \rightarrow \text{statement stmt-sequence}'$

$\text{stmt-sequence}' \rightarrow ; \text{statement stmt-sequence}' \mid \epsilon$

Non-deterministic rule

Non-deterministic rule is the rule defined to a multiple rules start with same factor and it have a rule to solve it to be deterministic.

Ex) $A \rightarrow \alpha B1 \mid \alpha B2 \mid \alpha B3$ **Solution:** $A \rightarrow \alpha A'$ $A' \rightarrow B1 \mid B2 \mid B3$

It may cause some problems of choose which part. So, it must be resolved to deterministic rule more clear.

Example in Tiny language:

$\text{If-stmt} \rightarrow \text{If exp then stmt-sequence end} \mid \text{If exp then stmt-sequence else stmt-sequence end}$

How to resolve it: $\text{If-stmt} \rightarrow \text{If exp then stmt-sequence If-stmt}'$

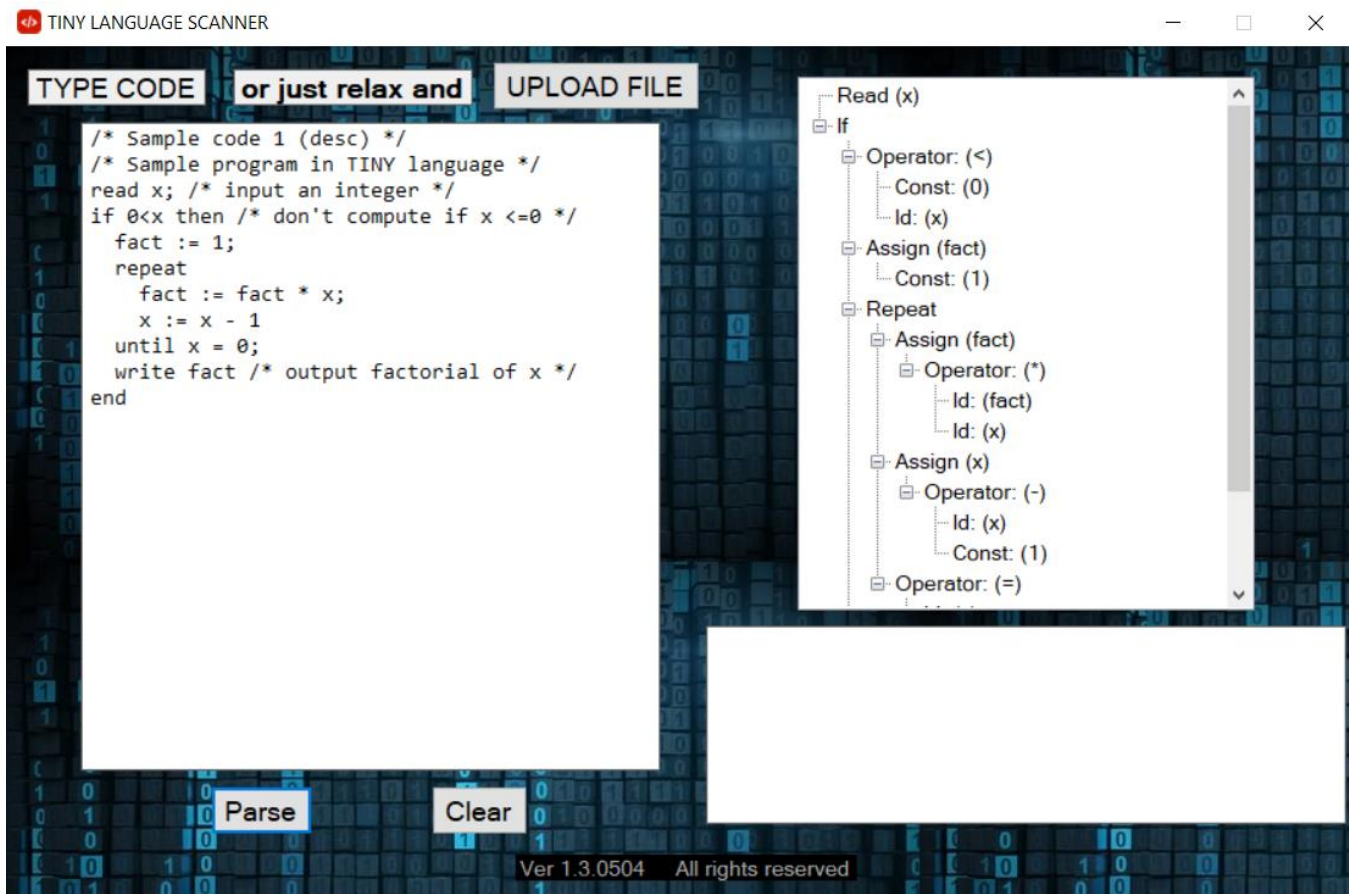
$\text{If-stmt}' \rightarrow \text{end} \mid \text{else stmt-sequence end}$

3.5. Emperimental Results

We will run some sample codes, and illustrate some snapshots of the resultant syntax tree.

Sample code I

```
/* Sample code 1 (desc) */
/* Sample program in TINY language */
read x; /* input an integer */
if 0<x then /* don't compute if x <=0 */
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact /* output factorial of x */
end
```



The screenshot displays the TINY LANGUAGE SCANNER application. On the left, the 'TYPE CODE' tab is active, showing the sample code. Below the code area are 'Parse' and 'Clear' buttons. On the right, the 'UPLOAD FILE' tab is active, showing a syntax tree diagram. The syntax tree is a hierarchical structure representing the code's logic. It starts with a 'Read (x)' node, followed by an 'If' node. The 'If' node has two children: 'Operator: (<)' and 'Const: (0)'. The 'Operator: (<)' node has a child 'Id: (x)'. The 'Const: (0)' node has a child 'Assign (fact)'. The 'Assign (fact)' node has a child 'Const: (1)'. The 'Const: (1)' node has a child 'Repeat' node. The 'Repeat' node has three children: 'Assign (fact)', 'Assign (x)', and 'Operator: (=)'. The 'Assign (fact)' node has two children: 'Operator: (*)' and 'Id: (fact)'. The 'Operator: (*)' node has two children: 'Id: (fact)' and 'Id: (x)'. The 'Assign (x)' node has two children: 'Operator: (-)' and 'Id: (x)'. The 'Operator: (-)' node has two children: 'Id: (x)' and 'Const: (1)'. The 'Operator: (=)' node has no children.

Figure 11 Parser Sample Code I

Sample code II

TINY LANGUAGE SCANNER

TYPE CODE or just relax and UPLOAD FILE

```

/*Sample code 2*/
/* GCD of two integer numbers */
read x;
read y;
min := 0-1;
if x < y then /* min of x & y */
  min := x
else
  min := y
end;
i := min;
repeat
  if ((x - i * (x / i) = 0)) then
    if (y - i * (y / i) = 0) then
      write i;
      i := 0
    else i := i - 1
    end
  else write 0-1
  end
until i > 0
  
```

Parse Clear

Ver 1.3.0504 All rights reserved

Repeat

- If
 - Operator: (=)
 - Operator: (-)
 - Id: (x)
 - Operator: (*)
 - Id: (i)
 - Operator: (/)
 - Id: (x)
 - Id: (i)
 - Const: (0)
 - If
 - Operator: (=)
 - Operator: (-)
 - Id: (y)
 - Operator: (*)
 - Id: (i)

Figure 12 Parser Sample Code II

Sample code III

TINY LANGUAGE SCANNER

TYPE CODE or just relax and UPLOAD FILE

```
/* Sample code 4 */
/* Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13
etc. */
read n; /* n is the number of elements */
i := 2;
if n > (i - 1) then
  n1 := 0;
  n2 := 1;
  write n1;
  write n2;
  repeat
    n3 := n1 + n2;
    write n3;
    n1 := n2;
    n2 := n3;
    i := i + 1
  until i > n
end
```

Parse Clear

Ver 1.3.0504 All rights reserved

Assign (n3)
Operator: (+)
Id: (n1)
Id: (n2)
Write
Id: (n3)
Assign (n1)
Id: (n2)
Assign (n2)
Id: (n3)
Assign (i)
Operator: (+)
Id: (i)
Const: (1)
Operator: (>)
Id: (i)
Id: (n)

Figure 13 Parser Sample Code III

Sample code with syntax errors

We will introduce some errors in the code and detect them with the parser showing the error type.

Errors introduced (Noted in the code):

- Missing semicolon: will give an unexpected token at the next one because the parser expected a semicolon at this place not any token else.
- Confuse (=) operator with the assign operator (:=)
- Use a statement that is not supported by the language grammar (like for)
- Use open parentheses without matching a closing one.

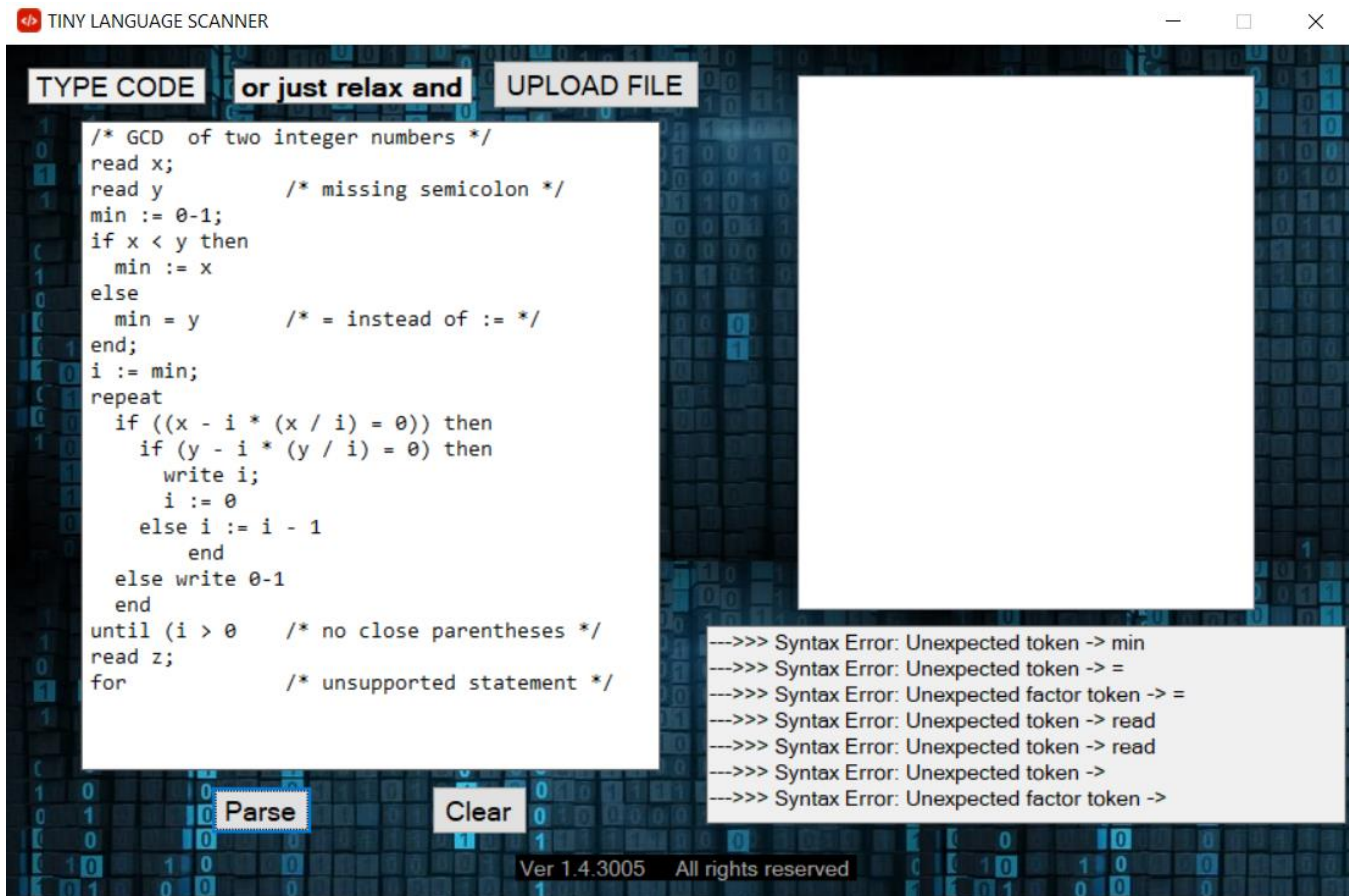


Figure 14 Parser Sample Code With Errors

REFERENCES

- [1] Louden, K. C. (1977). *Compiler Construction: Principles and Practice*. California: PWS Publishing Company.
- [2] *Introduction of Lexical Analysis*. (2020, May 28). Retrieved from GeeksforGeeks:
<https://www.geeksforgeeks.org/introduction-of-lexical-analysis/>
- [3] Terry, P. (2004). *Compiling with C# and Java*. Addison-Wesley.
- [4] *Lexical analysis*. (2020, May 9). Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Lexical_analysis
- [5] Appel, A. W. (1998). *Modern Compiler Implementation in C*. New York: Cambridge University Press.