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	**17P1023**
2	**17P6006**
3	**17P8229**
4	**17P3067**
5	**17P6085**

Parser Documentation

Table of Contents

List of Figures

1. Introduction

This report main purpose is to allow the reader to have a better understanding about the mechanism of the first two phases of any complier. We expect that the reader must be familiar with the basics of any programming language also he must know what is meant by regular expression, deterministic finite automaton(DFA) and non- deterministic finite automaton(NFA). At the beginning we will give a brief define what is a compiler, discuss some related concepts and definitions, after that will take a quick glance on its history then we are going to discuss both scanner and parser phases in details through the report. Compiler is a program that translates an executable program in one language (source program) into an executable program, usually in another language (target program) and it improves the program in some way. The compiler consists of six phases: Scanner, Parser, Semantic Analyzer, Source code optimizer, Code generator, and Target code optimizer; Our report illustrate only the first two phases. There is principle data structures used to allow communication between these phases such as tokens, syntax tree, symbol table, literal table, intermediate code, and temporary files. Owing to the fact that, scanner phase depends on tokens and parser phase depends on tokens and syntax tree so only those two data structures are briefly discussed through the rest of this report. Now let us look at some dates; The first compiler was written by Garce Hopper, in 1952 for the A-0 programming language. The first compiler was developed between 1954 and 1957. The related theories and algorithms were put in the 1960s and 1970s for example the classification of language (Chomsky hairearchy), The parsing problem was pursed (context-free language, parsing algorithms), the symbolic methods for expressing the structure of words of a programming language (finite automata, regular expressions), etc… Now let us shift for the recent advances in compiler design. More sophisticated algorithms for inferring and/or simplifying the information contained in program, window-based interactive Development Environment. Enough theoretical talking and dates and let us start exploring the compiler phases.

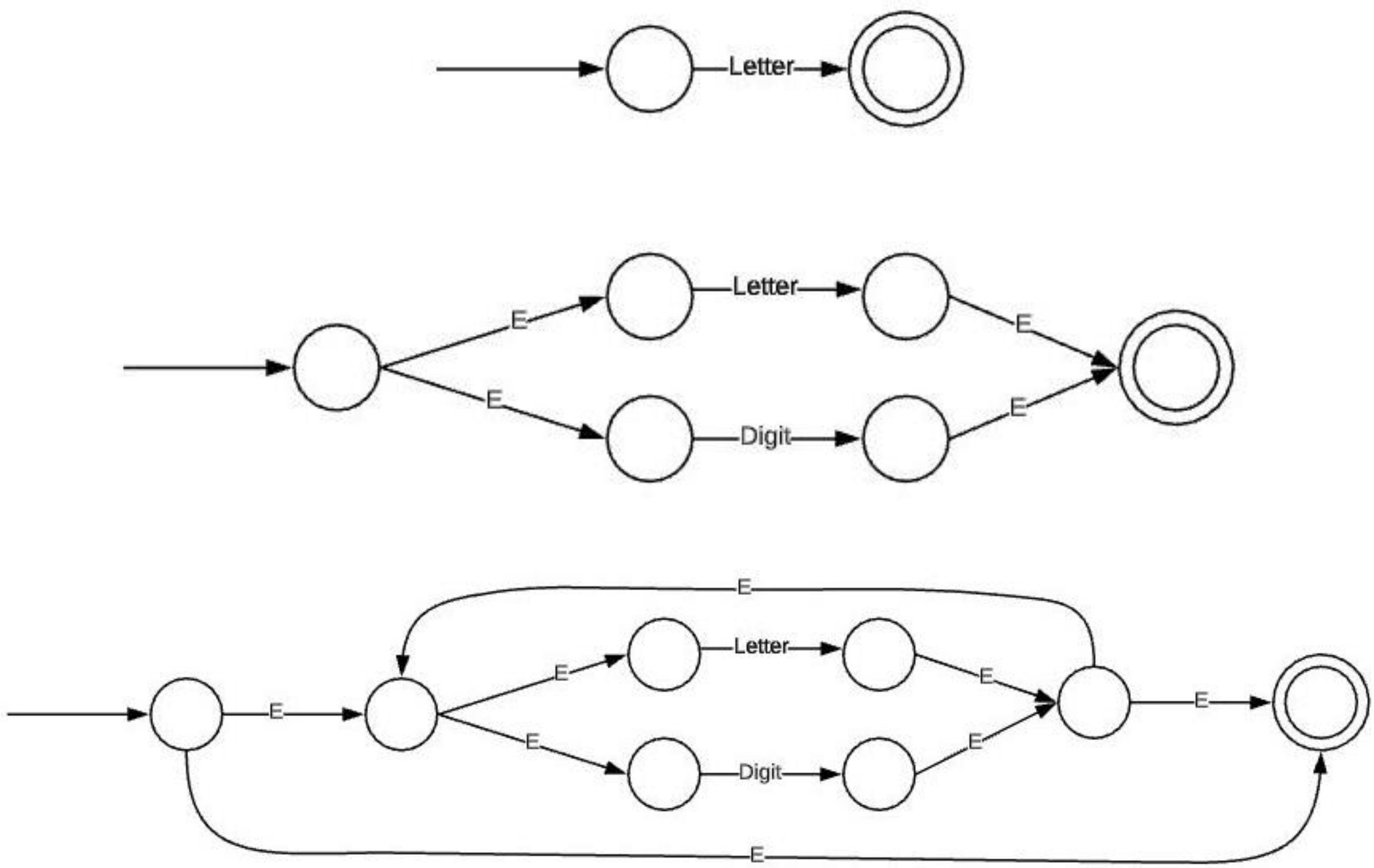
2. The Scanner Phase

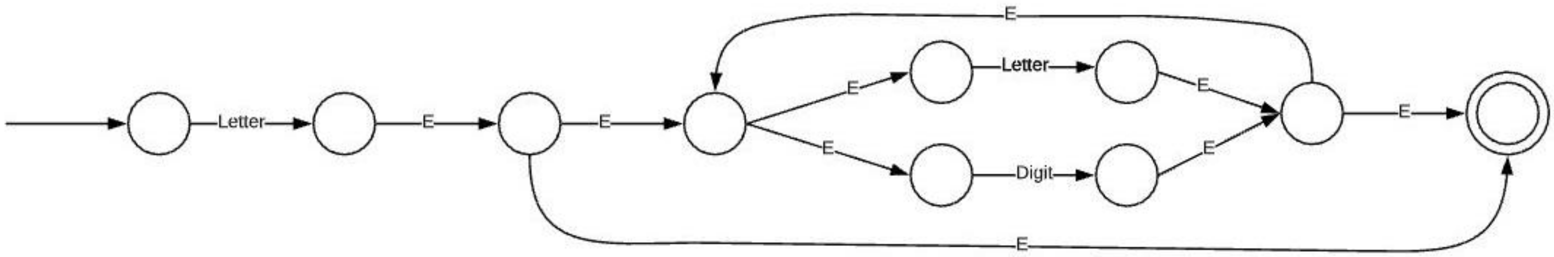
2.1 Scientific Background

- Scanner is a subroutine which is used by the compiler or we can consider it as a phase of the compilation process. This phase does the actual reading of the source program, which is in the form of a stream of characters. The scanner performs the lexical analysis: it collects sequences of characters into meaningful units called tokens.

First Regular expression:

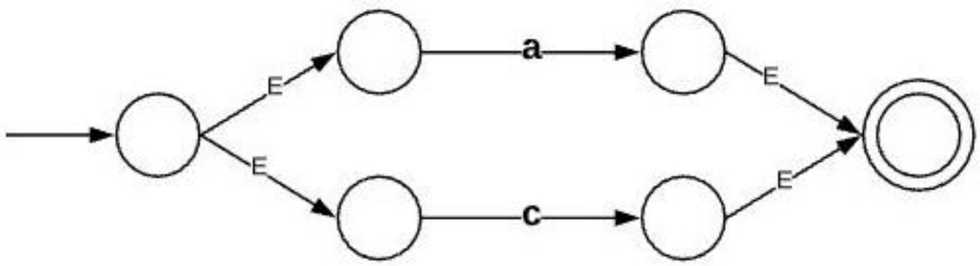
Letter(letter|Digit)*

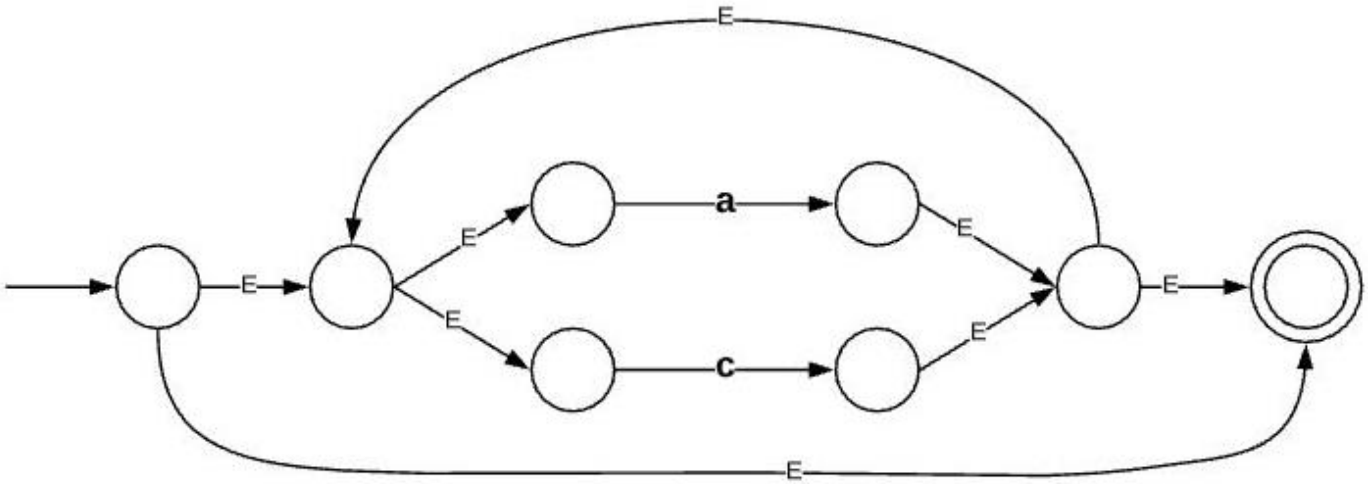


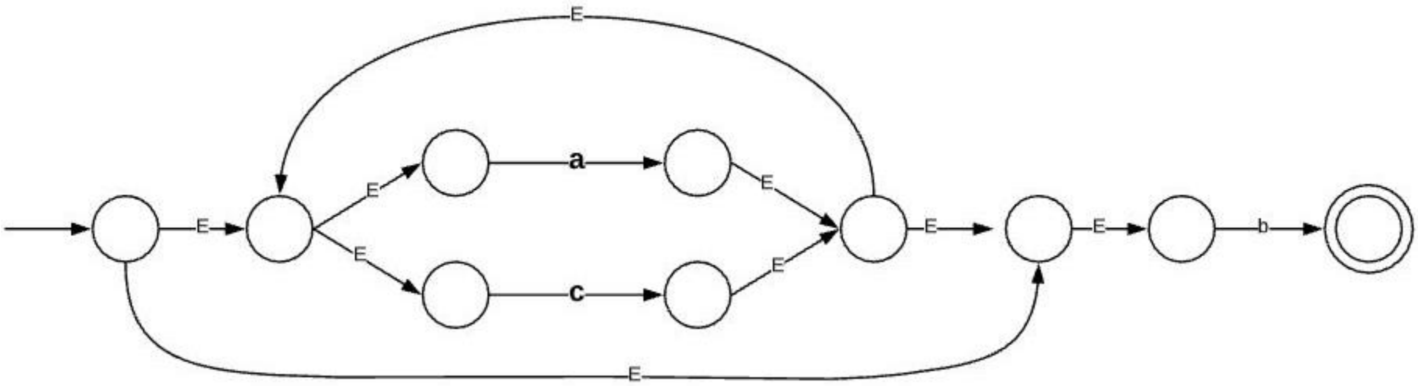


Second Regular expression:

(a|c)*b







2.2 Experimental Results

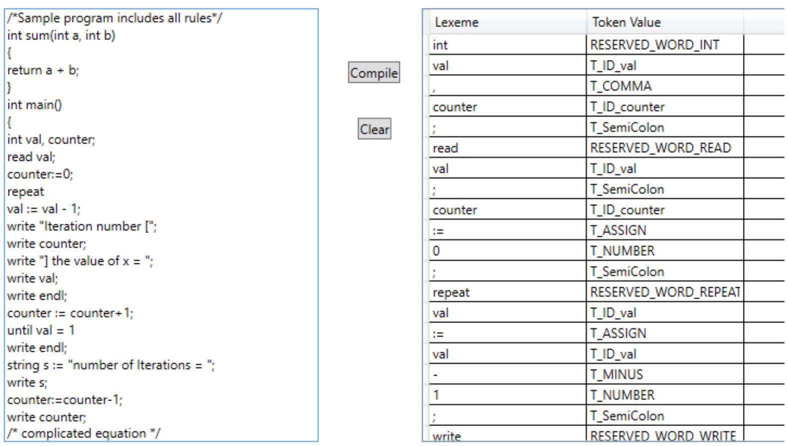
- Two **examples** showing the **input and output** of this phase and illustrate with screenshots.

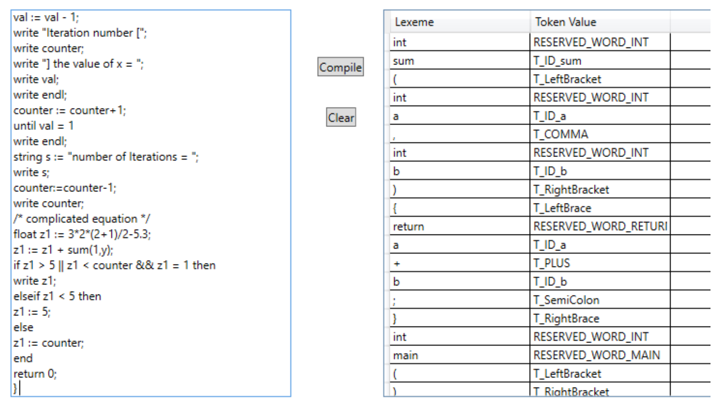
Example 1

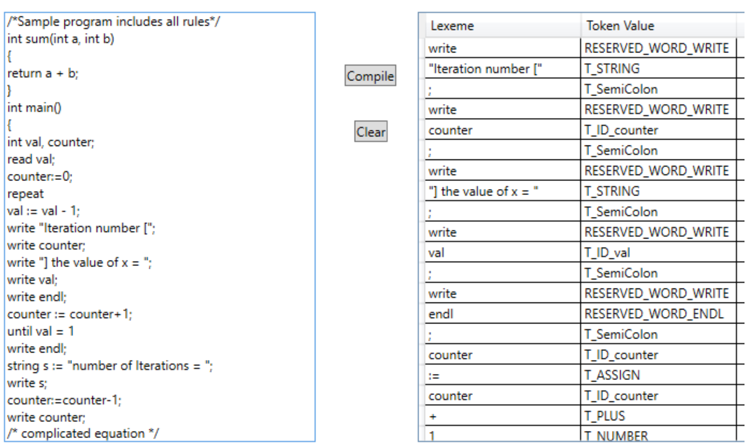
Input:

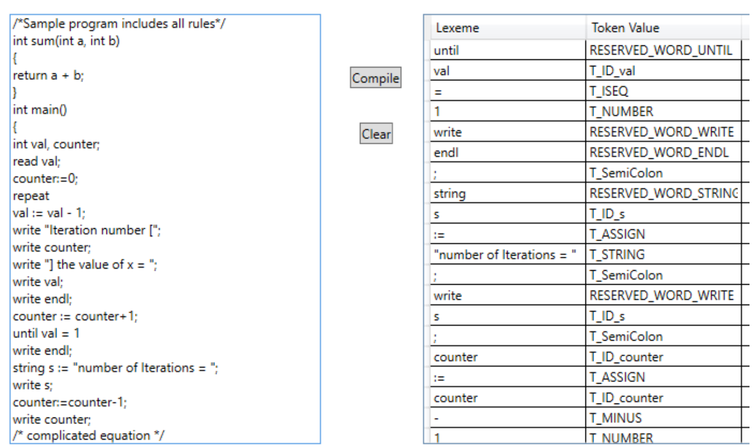
```
/*Sample program includes all rules*/
int sum(int a, int b)
{
return a + b;
}
int main()
{
int val, counter;
read val;
counter:=0;
repeat
val := val - 1;
write "Iteration number [";
write counter;
write "] the value of x = ";
write val;
write endl;
counter := counter+1;
until val = 1
write endl;
string s := "number of Iterations = ";
write s;
counter:=counter-1;
write counter;
/* complicated equation */
```

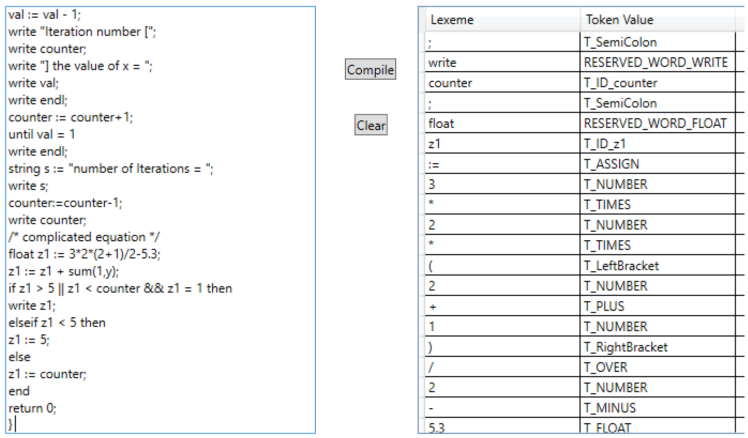
```
float z1 := 3*2*(2+1)/2-5.3;
z1 := z1 + sum(1,y);
if z1 > 5 || z1 < counter && z1 = 1 then
write z1;
elseif z1 < 5 then
z1 := 5;
else
z1 := counter;
end
return 0;
}
```



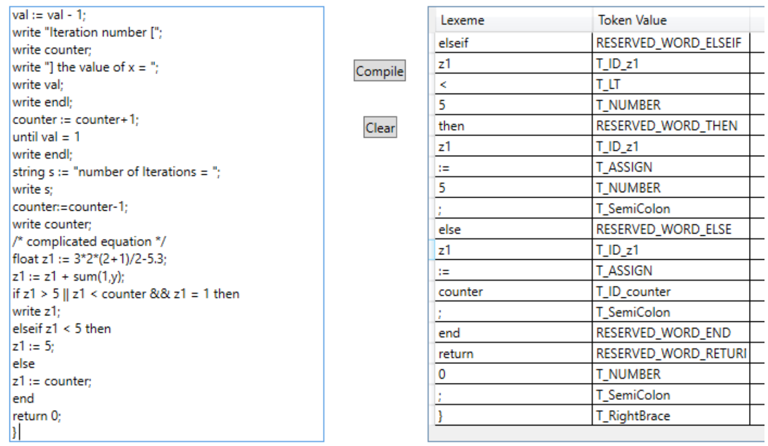




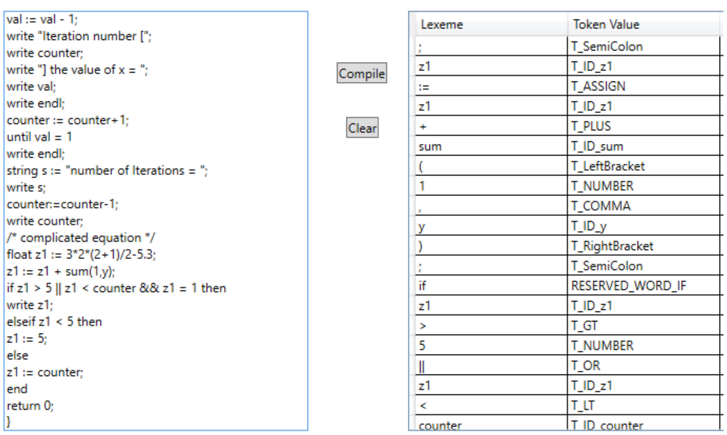




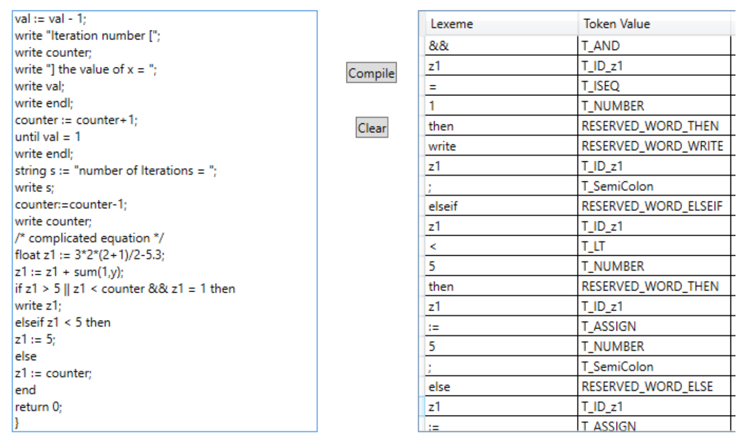
Lexeme	Token Value
;	T_SemiColon
write	RESERVED_WORD_WRITE
counter	T_ID_counter
;	T_SemiColon
float	RESERVED_WORD_FLOAT
z1	T_ID_z1
:=	T_ASSIGN
3	T_NUMBER
*	T_TIMES
2	T_NUMBER
*	T_TIMES
(	T_LeftBracket
2	T_NUMBER
+	T_PLUS
1	T_NUMBER
)	T_RightBracket
/	T_OVER
2	T_NUMBER
-	T_MINUS
5.3	T_FLOAT



Lexeme	Token Value
elseif	RESERVED_WORD_ELSEIF
z1	T_ID_z1
<	T_LT
5	T_NUMBER
then	RESERVED_WORD_THEN
z1	T_ID_z1
:=	T_ASSIGN
5	T_NUMBER
;	T_SemiColon
else	RESERVED_WORD_ELSE
z1	T_ID_z1
:=	T_ASSIGN
counter	T_ID_counter
;	T_SemiColon
end	RESERVED_WORD_END
return	RESERVED_WORD_RETURN
0	T_NUMBER
;	T_SemiColon
}	T_RightBrace



Lexeme	Token Value
;	T_SemiColon
z1	T_ID_z1
:=	T_ASSIGN
z1	T_ID_z1
+	T_PLUS
sum	T_ID_sum
(	T_LeftBracket
1	T_NUMBER
,	T_COMMA
y	T_ID_y
)	T_RightBracket
;	T_SemiColon
if	RESERVED_WORD_IF
z1	T_ID_z1
>	T_GT
5	T_NUMBER
||	T_OR
z1	T_ID_z1
<	T_LT
counter	T_ID_counter



Lexeme	Token Value
&&	T_AND
z1	T_ID_z1
=	T_ISEQ
1	T_NUMBER
then	RESERVED_WORD_THEN
write	RESERVED_WORD_WRITE
z1	T_ID_z1
;	T_SemiColon
elseif	RESERVED_WORD_ELSEIF
z1	T_ID_z1
<	T_LT
5	T_NUMBER
then	RESERVED_WORD_THEN
z1	T_ID_z1
:=	T_ASSIGN
5	T_NUMBER
;	T_SemiColon
else	RESERVED_WORD_ELSE
z1	T_ID_z1
:=	T_ASSIGN

Example 2

```
/* 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;
}
```

```
/* 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;
}
```

Compile

Clear

Lexeme	Token Value
int	RESERVED_WORD_INT
main	RESERVED_WORD_MAIN
(	T_LeftBracket
)	T_RightBracket
{	T_LeftBrace
int	RESERVED_WORD_INT
x	T_ID_x
;	T_SemiColon
read	RESERVED_WORD_READ
x	T_ID_x
;	T_SemiColon
if	RESERVED_WORD_IF
x	T_ID_x
>	T_GT
0	T_NUMBER
then	RESERVED_WORD_THEN
int	RESERVED_WORD_INT
fact	T_ID_fact
:=	T_ASSIGN
1	T_NUMBER

```
/* 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;
}
```

Compile

Clear

Lexeme	Token Value
1	T_NUMBER
;	T_SemiColon
repeat	RESERVED_WORD_REPEAT
fact	T_ID_fact
:=	T_ASSIGN
fact	T_ID_fact
*	T_TIMES
x	T_ID_x
;	T_SemiColon
x	T_ID_x
:=	T_ASSIGN
x	T_ID_x
–	Undefined_Symbol_ERROR
1	T_NUMBER
;	T_SemiColon
until	RESERVED_WORD_UNTIL
x	T_ID_x
=	T_ISEQ
0	T_NUMBER
write	RESERVED_WORD_WRITE

```
/* 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;
}
```

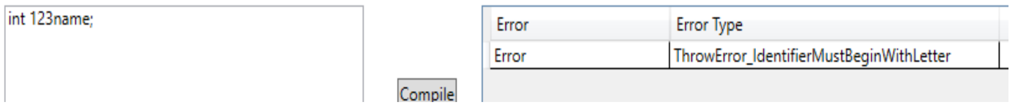
Compile

Clear

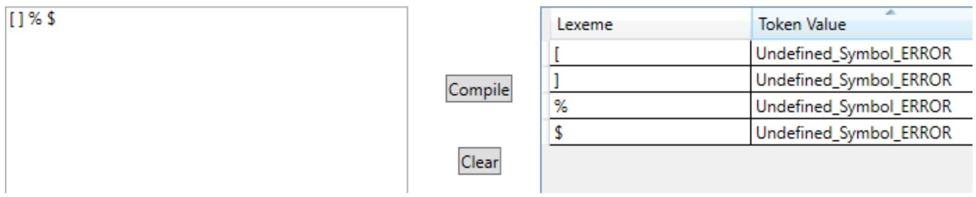
Lexeme	Token Value
;	T_SemiColon
x	T_ID_x
:=	T_ASSIGN
x	T_ID_x
–	Undefined_Symbol_ERROR
1	T_NUMBER
;	T_SemiColon
until	RESERVED_WORD_UNTIL
x	T_ID_x
=	T_ISEQ
0	T_NUMBER
write	RESERVED_WORD_WRITE
fact	T_ID_fact
;	T_SemiColon
end	RESERVED_WORD_END
return	RESERVED_WORD_RETURN
0	T_NUMBER
;	T_SemiColon
}	T_RightBrace

- Give **two examples of errors** appearing in this phase **and illustrate with screenshots**.

Error 1



Error2



3. The Parser Phase

3.1 Scientific Background

- This phase comes after lexical analysis and can be called Syntax Analysis or parsing . It is the task of the parser to determine the syntactic structure of a program from the tokens produced by the scanner and to construct a parse tree or syntax tree that represents this structure. The parsing step of the compiler reduces to a call to the parser.The syntax tree is defined as a dynamic data structure, in which each node consists of a record whose fields include the attributes needed for the remainder of the compilation process.

ENF:

- **A context-free Grammar for TINY:**

program → *stmt-sequence*

stmt-sequence → *stmt-sequence* ; *statement* | *statement*

statement → *if- stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*

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

repeat -stmt → **repeat** *stmt-sequence* **until** *exp*

assign -stmt → ***identifier*** **:=** *exp*

read -stmt → **read** ***identifier***

write -stmt → **write** *exp*

exp → *simple-exp comparison-op simple-exp* | *simple-exp*

comparison-op → < | =

simple-exp → *simple-exp addop term* | *term*

addop → + | -

term → *term mulop factor* | *factor*

mulop → * | /

factor → ***(****exp****)*** | ***number*** | ***identifier***

- EBNF:

program→ stmt-sequence

stmt-sequence → statement {; statement }

statement→ if- stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt

if -stmt → if exp then stmt-sequence [else stmt-sequence] end

repeat -stmt → repeat stmt-sequence until exp

read -stmt → read identifier

write -stmt → write exp

exp → simple-exp {comparison-op simple-exp } simple-exp → term {addop term}

addop → + | -

term → factor {mulop factor}

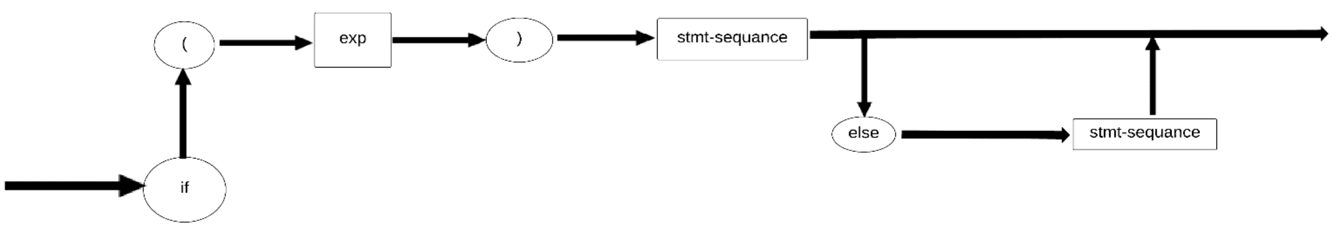
mulop → * | /

factor → (exp) | number | identifier

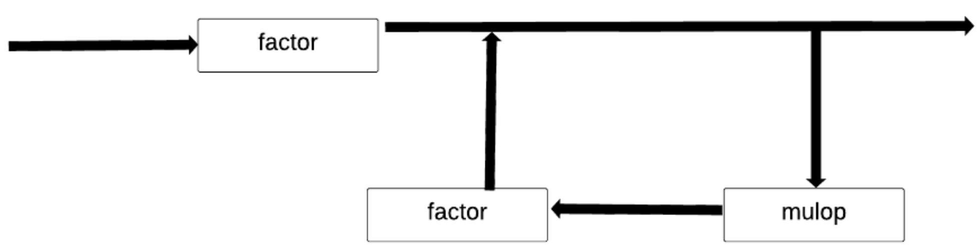
- Syntax Diagram :

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

if -stmt → if exp then stmt-sequence [else stmt-sequence] end



term → *term mulop factor* | *factor*



term → factor {mulop factor}

- **An example from the given grammar of a left recursive rule and how to resolve it.**

term → term mulop factor | factor

Solution

Term → factor Term'
Term' → ε | mulop factor Term'

- An example from the given grammar of a non-deterministic rule and how to resolve it.

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 **end** X
x → **end** | else stmt-sequence **end**

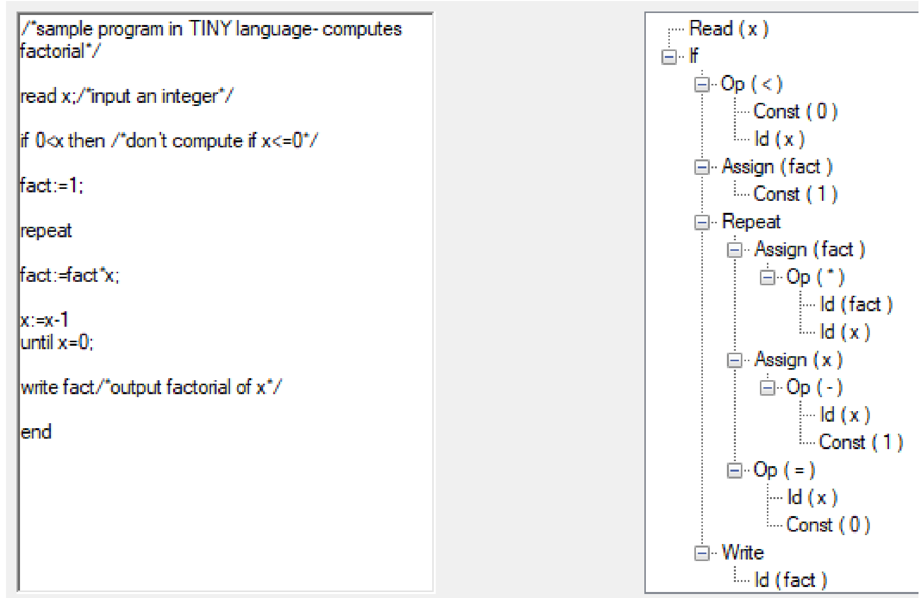
3.2 Experimental Results

- **Three examples** showing the **input and output** of this phase, **illustrating** with screenshots.

Sample1

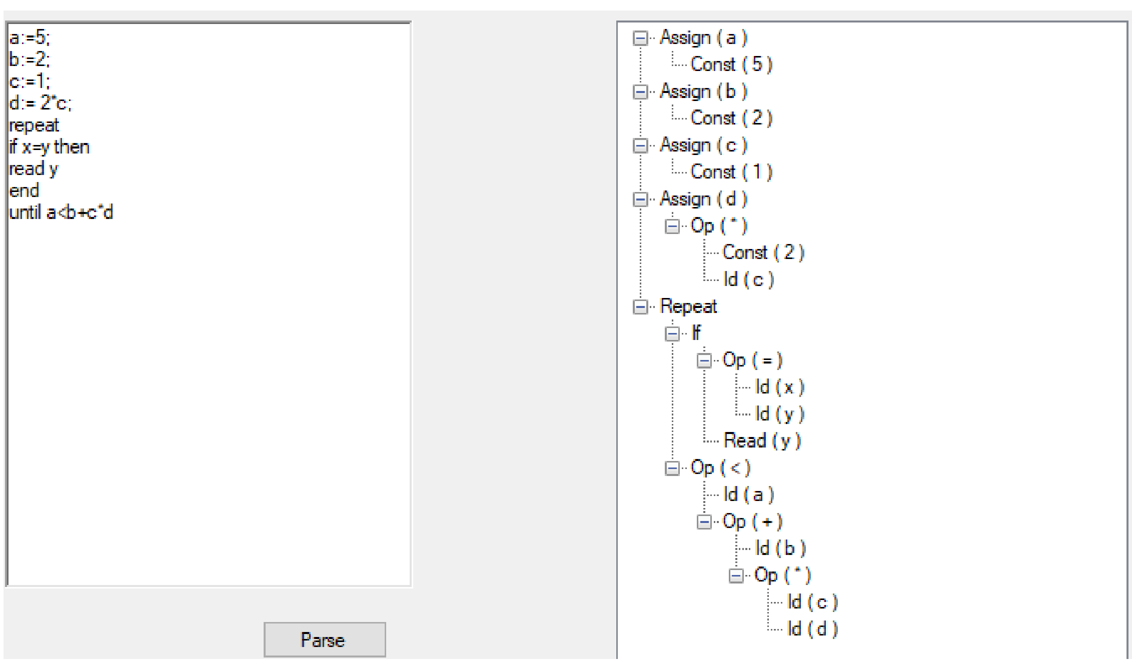
/*Sample program in TINY language – computes factorial*/

```
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
```



Sample2

```
 a:=5;
b:=2;
c:=1;
d:= 2*c;
repeat
if x=y then
 read y
end
until a<b+c*d
```



Sample3

a:=5;

d:=3+5;

if a < 10 then

c:=d

end;

x := (2+3)*4

- **Three examples of errors** appearing in this phase, **illustrating with screenshots**

Error1

/*untill is written wrong*/

repeat

stmt:=y

untill a<b

Error2

write x write y

Error3

/* no repeat */

x:= 3+5;

until x = 3;

List of References

- Compiler Construction Principles and Practice, Kenneth C. Louden, 1998
- Compilers: Principles, Techniques, and Tools ,by Monica S. Lam, R. Sethi, Jeffrey D. Ullman, A.V.Aho,2013.