Compiler Project

BCSE III - Compiler Design Lab
Group 1
Project 1

Problem Statement

Designing **CFG, lexical Analyser, Parser and Symbol Table** for a C like language with the following language constructs.

1. Data Types integer, real character.

2. Declaration Statements : identifiers are declared in declaration statements as basic data types and may also be assigned constant values.

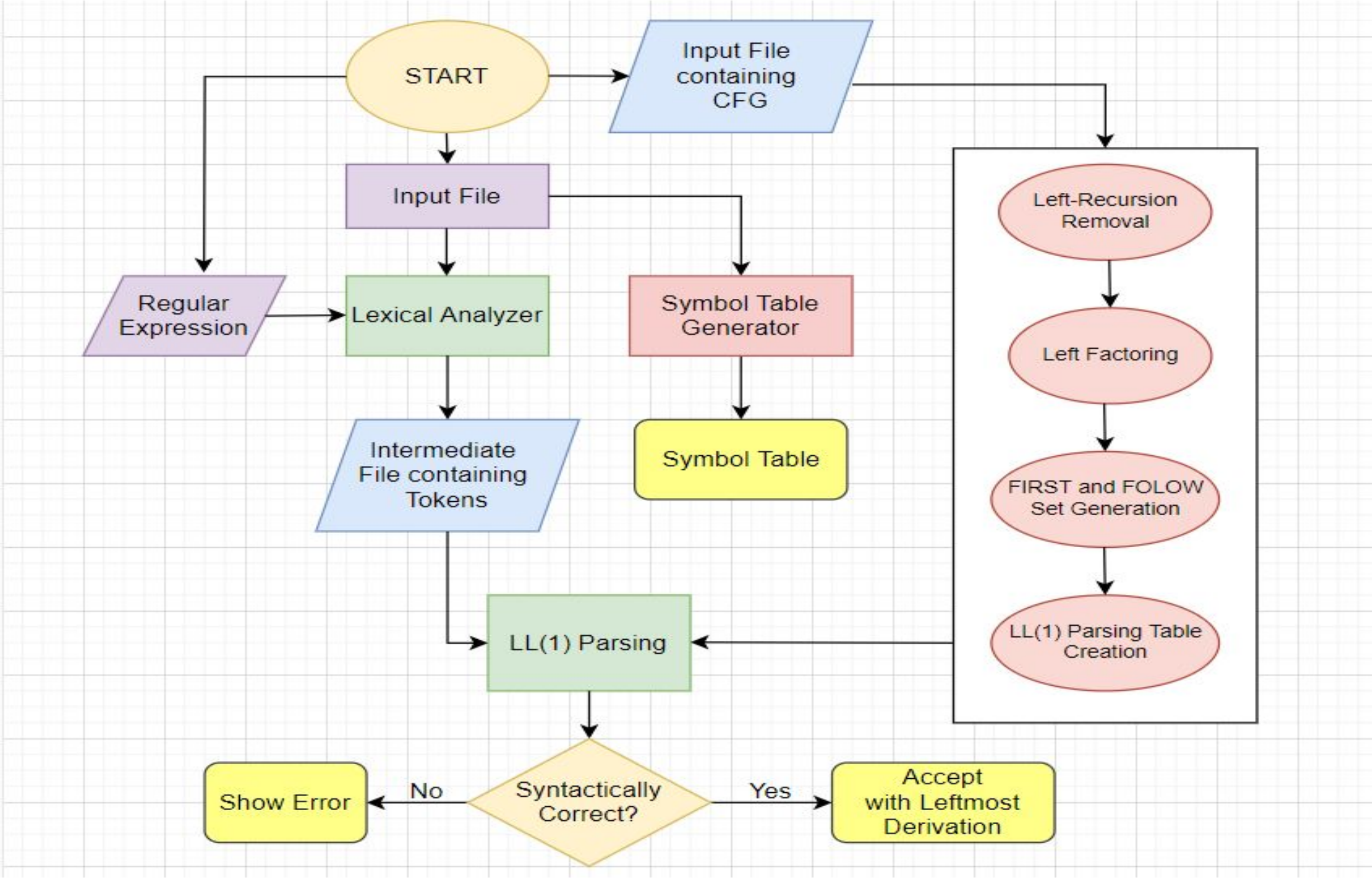
3. Condition Constructs :- if, then, else.

4. Relational Operators used in the if statement like < , >, == , !=.

5. Nested Statements are supported , there may be if statement without else statement and assignments to variable using I/O constructs can be present in the code like :- cin >> x (read into x) , cout << x (write the value of x to output).

6. Constraints :- Only function is main and there is no other function. The main function does not contain arguments and no return statements.

Flow Diagram:



Part 1 (Lexical Analyser)

The intention is to create a lexical analyser that converts the given code into a stream of tokens that can be parsed by the parse against a valid Context Free Grammar.

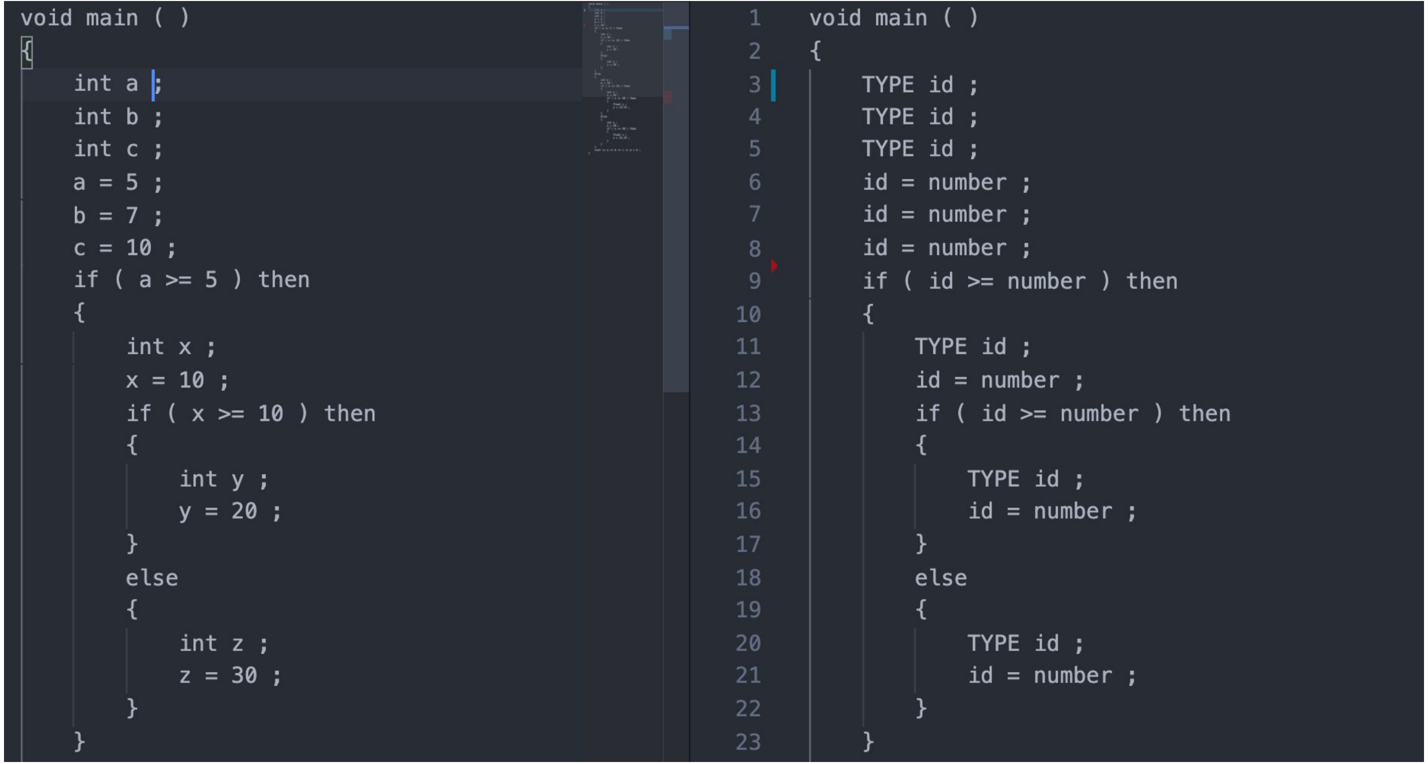
Key Points of the Design

1. Regular expression that can be matched with the given code to convert the keywords, constants, variables and types into tokens.

2. Implemented using the regex library of c++.

3. Implemented RE to DFA to understand the internal working of lexical analyser.

Proof Of Work (Lexical Analyser)

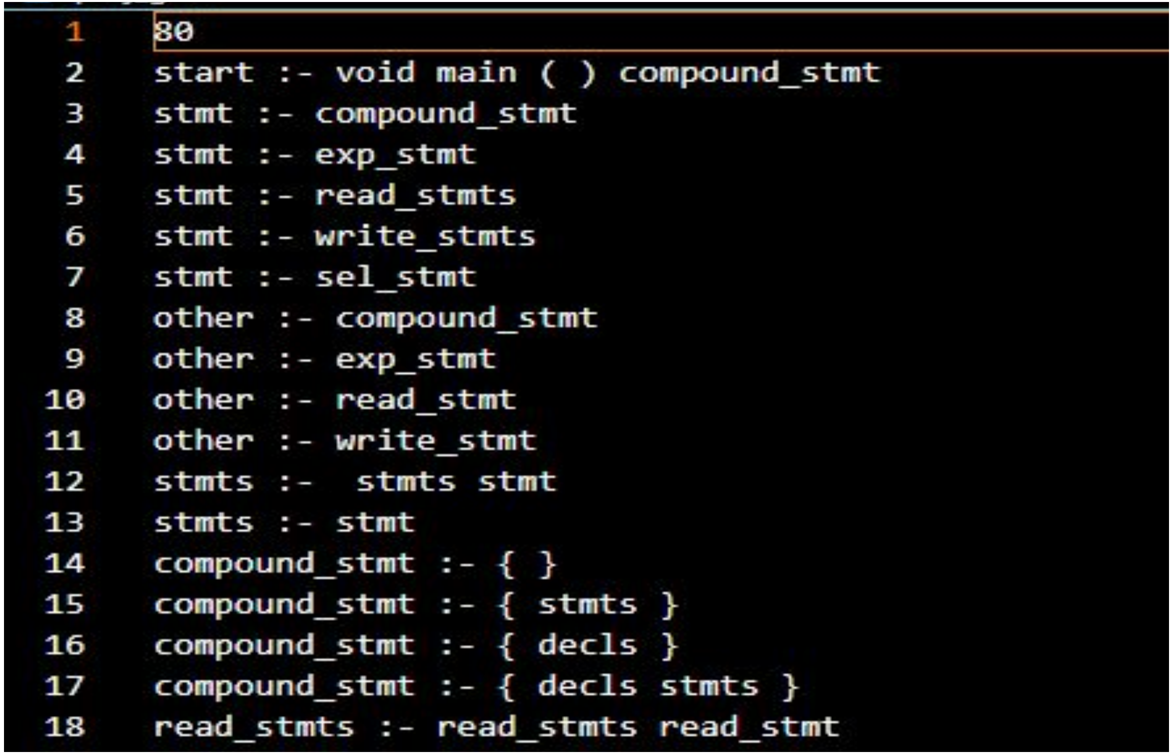


```
void main ( )
{
    int a ;
    int b ;
    int c ;
    a = 5 ;
    b = 7 ;
    c = 10 ;
    if ( a >= 5 ) then
    {
        int x ;
        x = 10 ;
        if ( x >= 10 ) then
        {
            int y ;
            y = 20 ;
        }
        else
        {
            int z ;
            z = 30 ;
        }
    }
```

```
void main ( )
{
    TYPE id ;
    TYPE id ;
    TYPE id ;
    id = number ;
    id = number ;
    id = number ;
    if ( id >= number ) then
    {
        TYPE id ;
        id = number ;
        if ( id >= number ) then
        {
            TYPE id ;
            id = number ;
        }
        else
        {
            TYPE id ;
            id = number ;
        }
    }
```

LHS of the image shows the input file (a sample code with the given constructs) and the RHS contains the code converted into stream of tokens that can be parsed by the LL1 Parser.

Part 2 (CFG Designing)



```
80
start :- void main ( ) compound_stmt
stmt :- compound_stmt
stmt :- exp_stmt
stmt :- read_stmts
stmt :- write_stmts
stmt :- sel_stmt
other :- compound_stmt
other :- exp_stmt
other :- read_stmt
other :- write_stmt
stmts :-  stmts stmt
stmts :- stmt
compound_stmt :- { }
compound_stmt :- { stmts }
compound_stmt :- { decls }
compound_stmt :- { decls stmts }
read_stmts :- read_stmts read_stmt
```

start symbol :- {start}

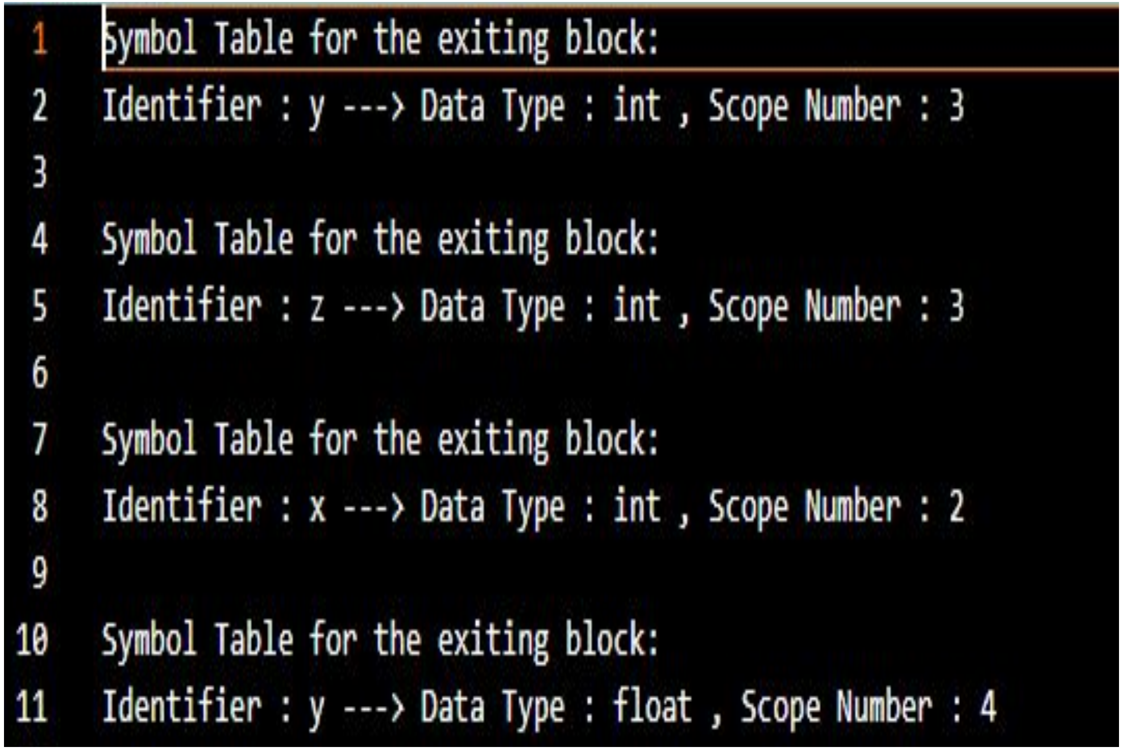
Non terminal symbols :- {start, stmt, other, stmts, compound_stmt, read_stmts, read_stmt, write_stmt, write_stmts, inputs, outputs, INP_OP, OUT_OP, decls, init_decl_list, …}

Terminal symbols :- {id, || , cin, cout, +, -, real, number, &&, *, /, %, int, float, char, <<, >>, (,), {, }, =, ==, ;, >=, <=, !=, ….}

Productions

start :- void main ( ) compound_stmt
stmt :- compound_stmt
stmt :- exp_stmt
stmt :- read_stmts
stmt :- write_stmts
stmt :- sel_stmt
other :- compound_stmt
other :- exp_stmt
other :- read_stmt
other :- write_stmt
stmts :- stmts stmt
stmts :- stmt
compound_stmt :- { }
compound_stmt :- { stmts }
compound_stmt :- { decls }
compound_stmt :- { decls stmts }

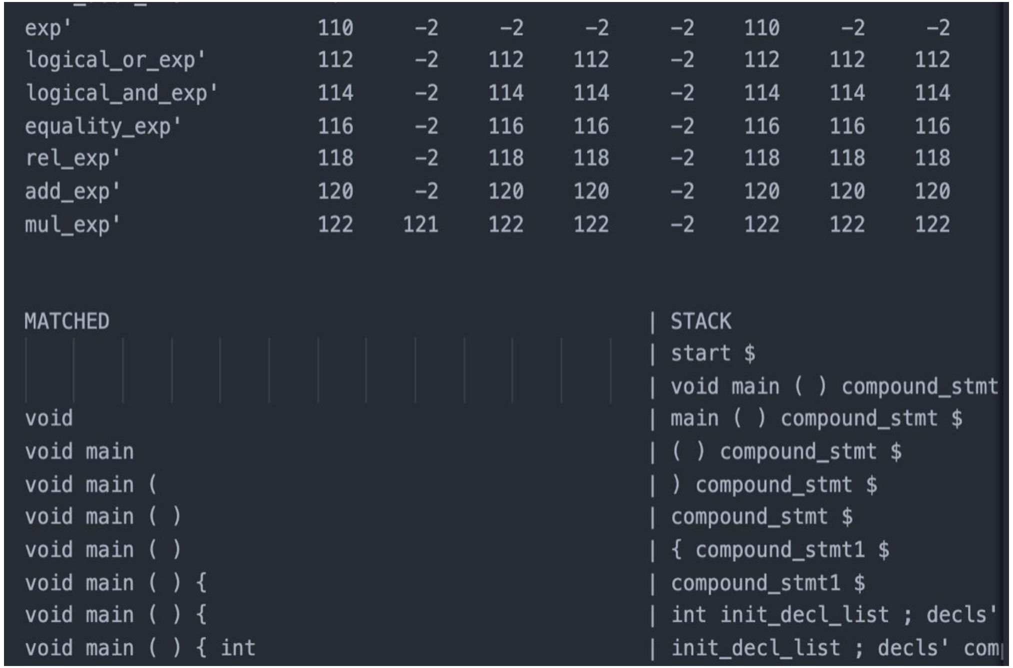
Part 3 (Symbol Table Design)



```
Symbol Table for the exiting block:
Identifier : y ---> Data Type : int , Scope Number : 3

Symbol Table for the exiting block:
Identifier : z ---> Data Type : int , Scope Number : 3

Symbol Table for the exiting block:
Identifier : x ---> Data Type : int , Scope Number : 2

Symbol Table for the exiting block:
Identifier : y ---> Data Type : float , Scope Number : 4
```

- Designed Symbol table using multiple hash tables, implemented as a list of hash tables.
- One hash table is created for each visible scope.
- Whenever a token is inserted in the symbol table, the lexeme value of the **id** is the key and the value consists of attributes: data type of the **id** and scope number for the **id**.
- While scanning the C program, whenever it encounters a new block of code, a new hash table is generated in the front of the list and when it exits the block, the hash table in the front of the list is deleted.

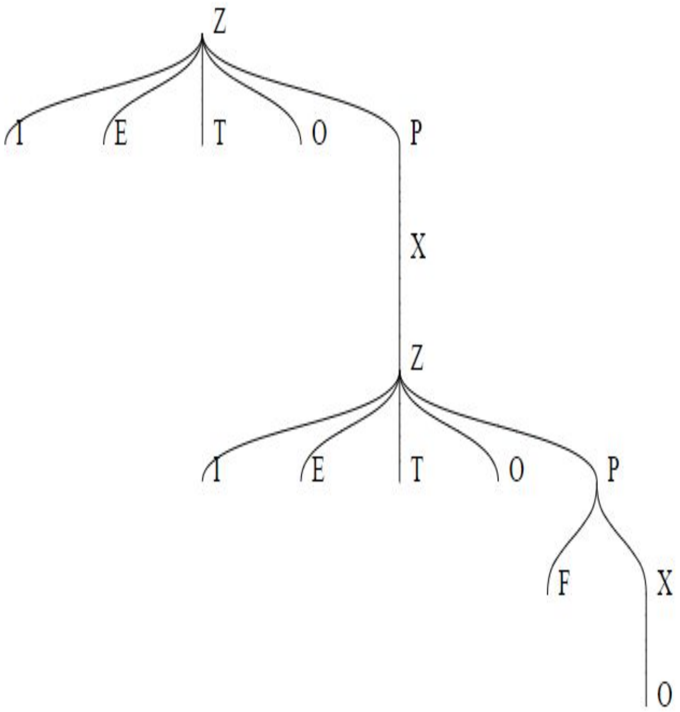
Part 4 (Syntax Analysis-LL(1) Parsing)



```
exp'                 110    -2    -2    -2    -2   110    -2    -2
logical_or_exp'      112    -2   112   112    -2   112   112   112
logical_and_exp'     114    -2   114   114    -2   114   114   114
equality_exp'        116    -2   116   116    -2   116   116   116
rel_exp'             118    -2   118   118    -2   118   118   118
add_exp'             120    -2   120   120    -2   120   120   120
mul_exp'             122   121   122   122    -2   122   122   122


MATCHED                                     | STACK
                                            | start $
                                            | void main ( ) compound_stmt
void                                        | main ( ) compound_stmt $
void main                                   | ( ) compound_stmt $
void main (                                 | ) compound_stmt $
void main ( )                               | compound_stmt $
void main ( )                               | { compound_stmt1 $
void main ( ) {                             | compound_stmt1 $
void main ( ) {                             | int init_decl_list ; decls'
void main ( ) { int                         | init_decl_list ; decls' com
```

After Lexical Analysis phase, the tokens are generated. These code structure in the “token” format is stored in “lex_output.txt” file. The CFG which is stored in the “proj_gr.txt” file along with the Lexical Analyzer output is fed into the “LL1.cpp” as an input. The “LL1.cpp” file will remove Left recursion from the provided grammars, then Left factor the improved form. After that the LL1 Parsing table will be framed and filled up with their values. Then the input sentence will be parsed using a Stack data structure. The corresponding parsing data will be stored in “final_output.txt” .

Parse Tree Generation Example 1:



CFG:

Z ::= I E T O P

P ::= F X

P ::= X

P ::= #

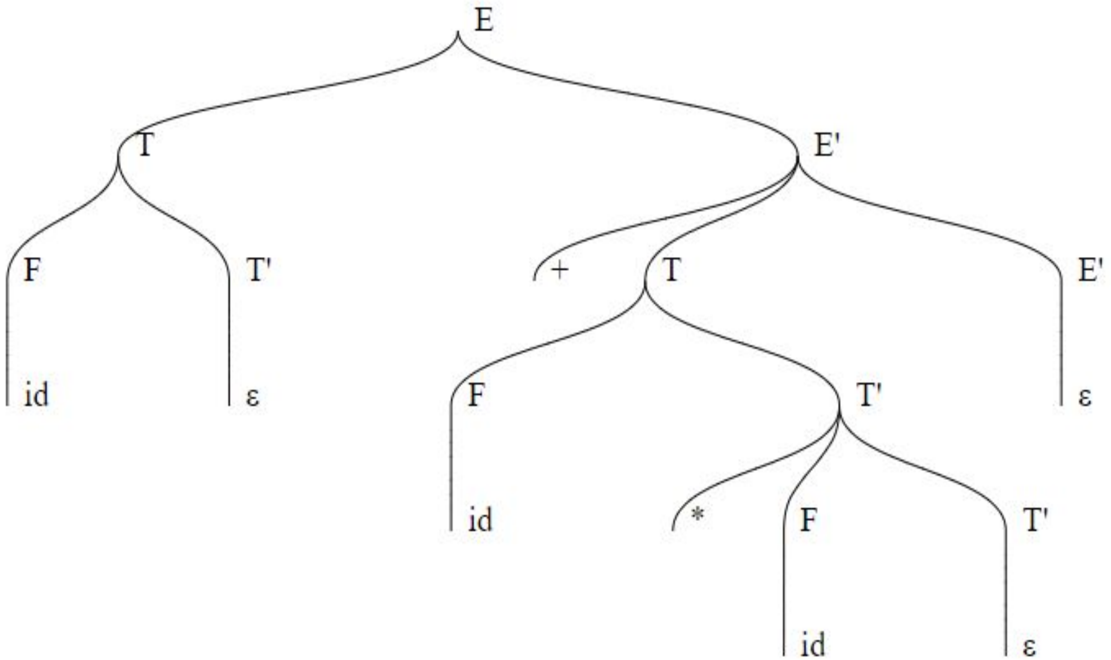
X ::= O

X ::= Z

Token Stream:

I E T O I E T O F O

Parse Tree Generation Example 2:



CFG:

E ::= T E'

E' ::= + T E'

E' ::= #

T ::= F T'

T' ::= * F T'

T' ::= ''

F ::= (E)

F ::= id

Token Stream:

id + id * id

Thanks!

Team Members

1. Agniv Ghosh 001910501086

2. Hrishikesh Mallick 001910501085

3. Debargha Mukherjee 001910501067

4. Trishit Mukherjee 001910501089