Compiler Design

PROJECT 2

Team Members

- Arka Choudhuri - 63
- Avigyan Bhattacharya -71
- Anirban Das - 77
- Ratul Chakraborty - 88

Problem Statement

Consider a simple C-like language with:

Data Types: integer, real and character

Declaration statements: identifiers are declared in declaration statements as basic data types and may also be assigned constant values (integer of floating)

Condition constructs: if, then, else. Relational operators used in the if statement are < (less than), > (greater than), == (equal) and != (not equal). Example:

```
if(a<10) then a=a*a; else a-a/2;
```

Nested statements are supported. There may be if statement without else statement.

Assignments to the variables are performed using the input/output constructs:

```
cin >> x - Read into variable x.

cout << x - Write variable x to output.
```

Arithmetic operators ... %) and assignment operator = are supported

Only function is main(), there is no other function. The main() function does not contain arguments and no return statements.

Part I - Construct a CFG for this language.

Part II - Write a lexical analyser to scan the stream of characters from a program written in the above language and generate stream of tokens.

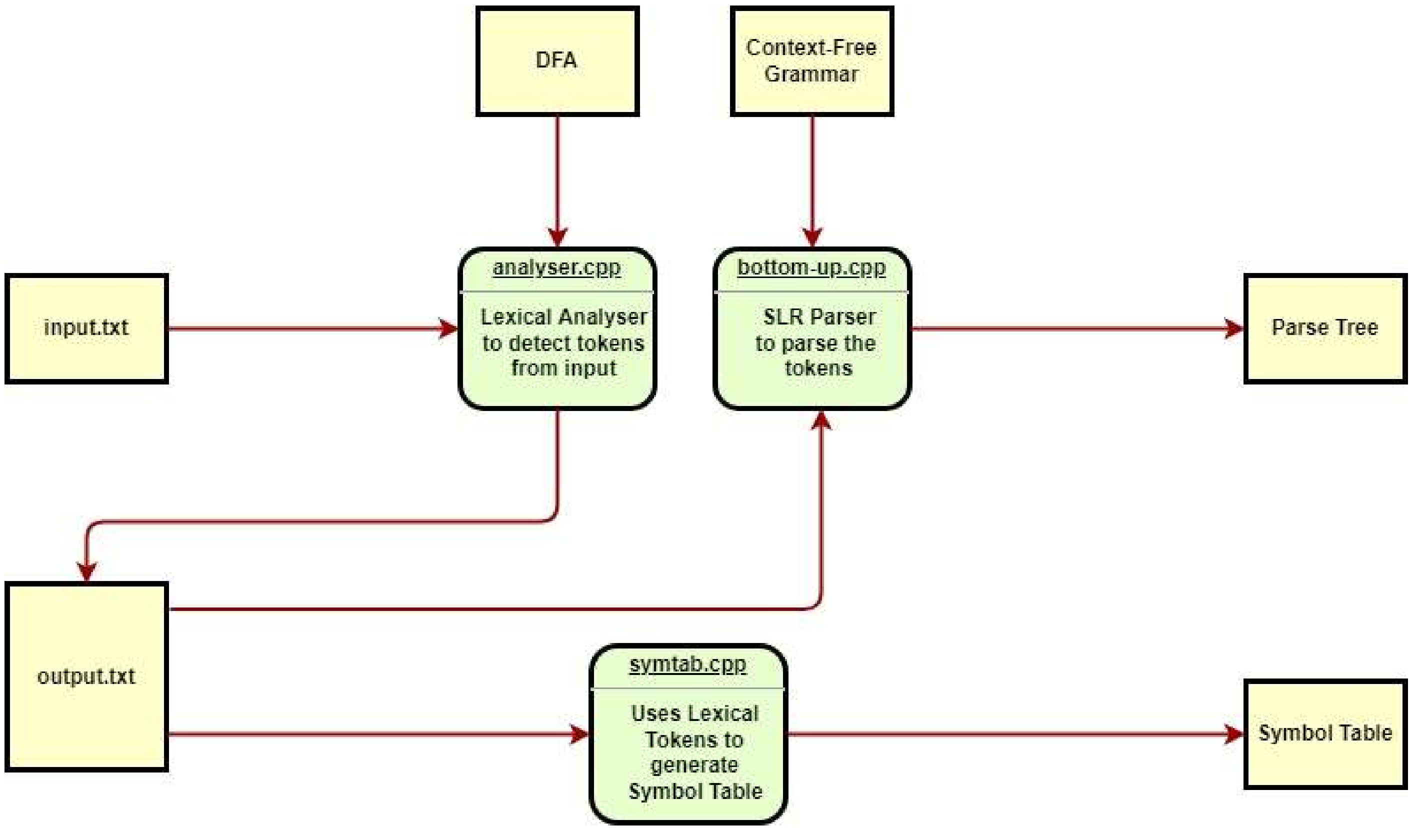
Part III Maintain a symbol table with appropriate data structures.

Part IV Write a bottom-up parser for this language (modules include Item-set construction, computation of FOLLOW, parsing table construction and parsing).

File Tree Structure

```
├── DFA
 - Lexical Analyzer implementation to extract, detect and classify tokens
  ├── analyser.cpp
   - main function giving the output
  ├── dfa.h
  ├── dfa_initiator.h
   - helper header files for dfa
  ├── input.txt
   - input code of C-like language
  ├── output.txt
   - output of all the tokens

├── SYMBOLTAB
  ├── symtab.cpp
    - Symbol table implementation using multiple hash tables   as data structure to store
identifiers
  ├── symbol_table.txt
    - Output of symbol table scope wise showing name, type and line no of the variables

├── PARSING
  ├── bottom-up.cpp
 - SLR parser implementation involving constructing the parsing tree and item sets
  ├── graph.pdf
   - Parse tree output
  ├── input.txt
   - Productions of our grammar
  ├── output.txt
   - Item sets output

├── compile_unix.sh
├── compile_windows.sh
 - Compiles entire file tree and gives output one by one to respective output text files –
separate for windows and unix environment
```

Overall Flow Diagram


DFA
Context-Free Grammar
input.txt
analyser.cpp
Lexical Analyser to detect tokens from input
bottom-up.cpp
SLR Parser to parse the tokens
Parse Tree
output.txt
symtab.cpp
Uses Lexical Tokens to generate Symbol Table
Symbol Table

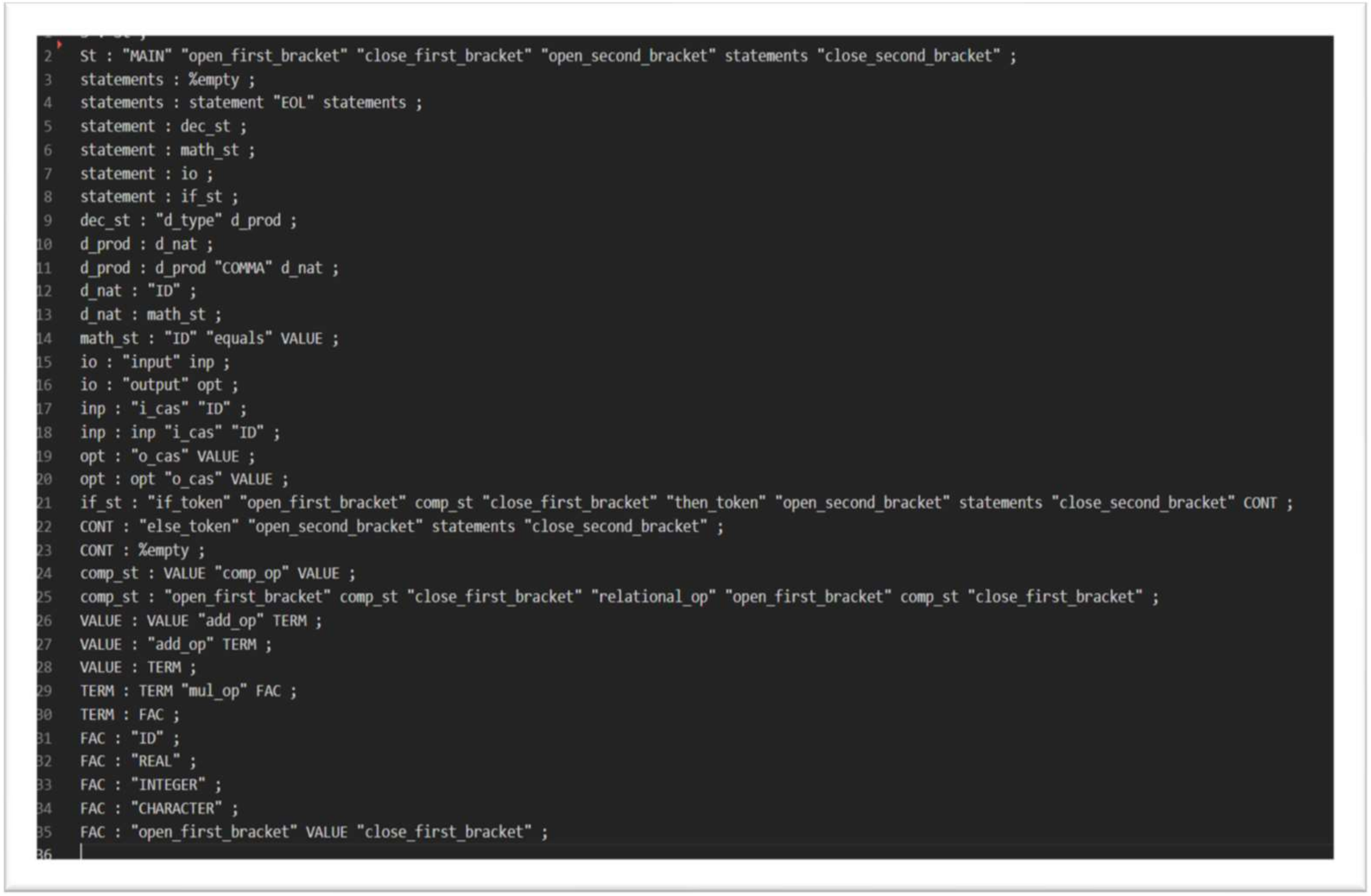
Context - Free Grammar

- Grammar supports all the features mentioned in the problem statement.
- Every program is must start with a main.
- The main contains no arguments and doesn't return anything.
- Int, real and char are the supported data types.
- Arithmetic operators: +,-,*,% are supported along with assignment operator: =
- Nested if-else statements are supported.
- If statements don't need to followed by an else statement.
- Relational operators: <, >, == ,!= are supported in if statement

Nonterminals - S, St, statements, statement, dec_st, math_st, io, if_st, d_prod, d_nat, VALUE, inp, opt, comp_st, CONT, TERM, FAC

Terminals- main, (,), {, }, ε, ;, int, char, real, ,, ID, =, cin, cout, <<,>>, if, then, else, <, >, <=, >=, !=, ==, and, or, +, -, *, /, %, real_number, integer_number, character

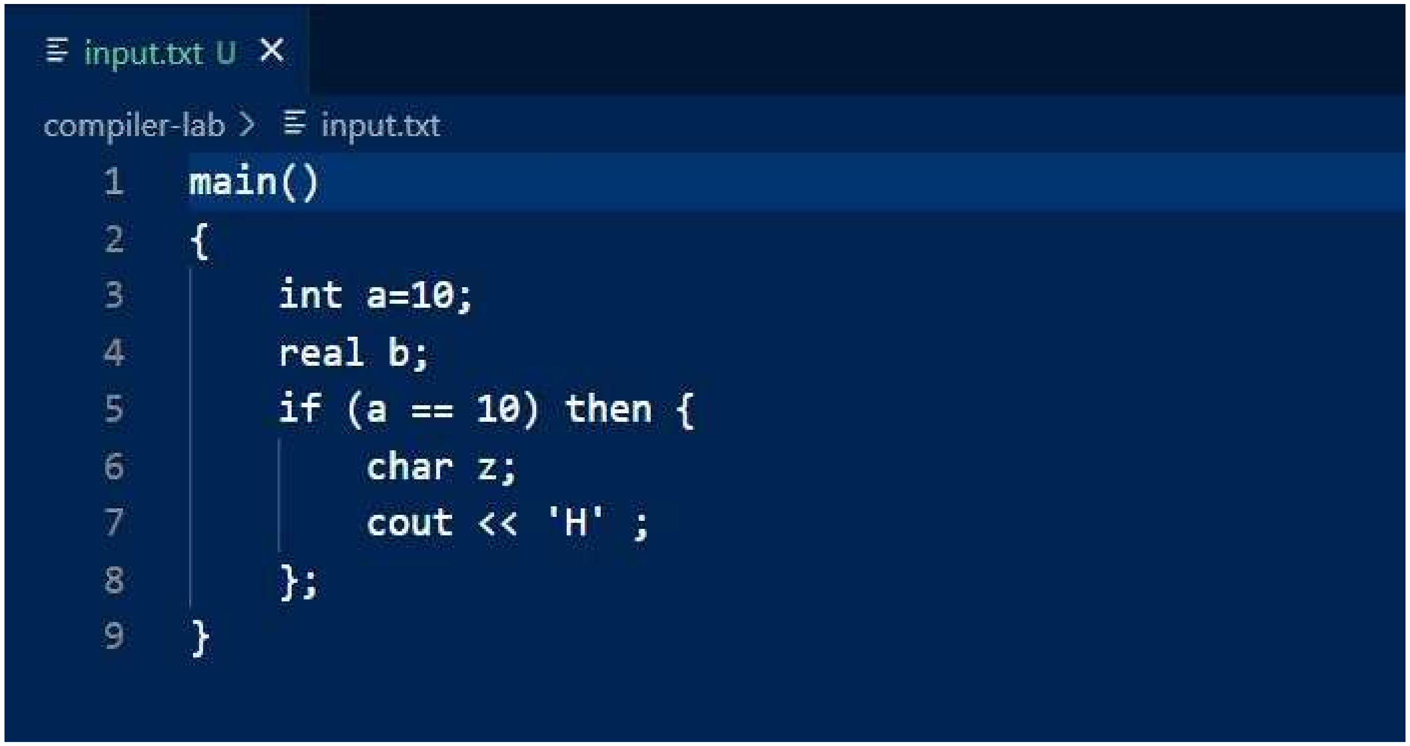
Context - Free Grammar (Productions)



```
St : "MAIN" "open_first_bracket" "close_first_bracket" "open_second_bracket" statements "close_second_bracket" ;
statements : %empty ;
statements : statement "EOL" statements ;
statement : dec_st ;
statement : math_st ;
statement : io ;
statement : if_st ;
dec_st : "d_type" d_prod ;
d_prod : d_nat ;
d_prod : d_prod "COMMA" d_nat ;
d_nat : "ID" ;
d_nat : math_st ;
math_st : "ID" "equals" VALUE ;
io : "input" inp ;
io : "output" opt ;
inp : "i_cas" "ID" ;
inp : inp "i_cas" "ID" ;
opt : "o_cas" VALUE ;
opt : opt "o_cas" VALUE ;
if_st : "if_token" "open_first_bracket" comp_st "close_first_bracket" "then_token" "open_second_bracket" statements "close_second_bracket" CONT ;
CONT : "else_token" "open_second_bracket" statements "close_second_bracket" ;
CONT : %empty ;
comp_st : VALUE "comp_op" VALUE ;
comp_st : "open_first_bracket" comp_st "close_first_bracket" "relational_op" "open_first_bracket" comp_st "close_first_bracket" ;
VALUE : VALUE "add_op" TERM ;
VALUE : "add_op" TERM ;
VALUE : TERM ;
TERM : TERM "mul_op" FAC ;
TERM : FAC ;
FAC : "ID" ;
FAC : "REAL" ;
FAC : "INTEGER" ;
FAC : "CHARACTER" ;
FAC : "open_first_bracket" VALUE "close_first_bracket" ;
```

Sample Input Code

We demonstrate our compiler components and their working using a simple code in the C-like language as instructed



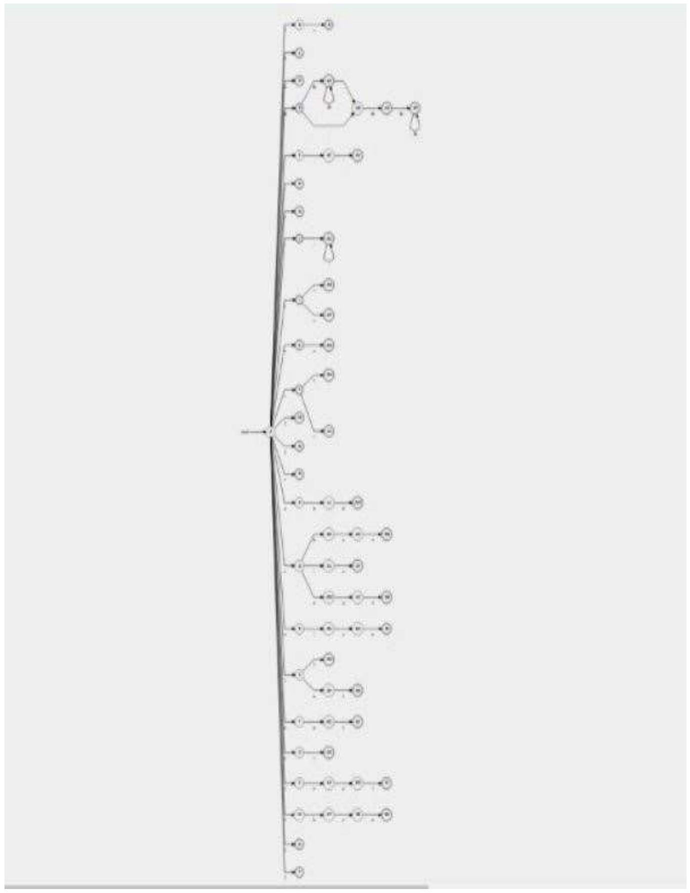
```
main()
{
    int a=10;
    real b;
    if (a == 10) then {
        char z;
        cout << 'H' ;
    };
}
```

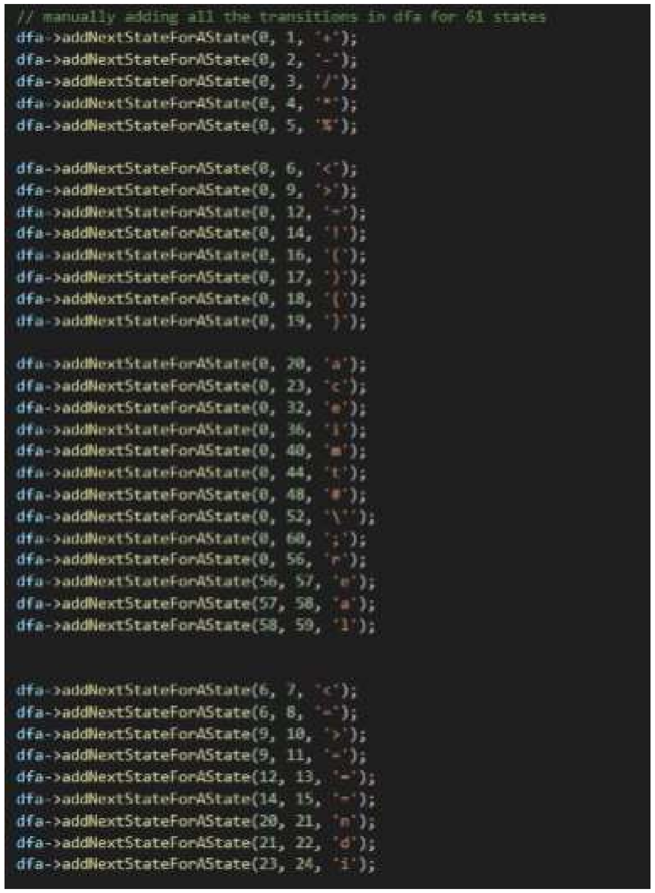
Lexical Analyser

- A 62 - state DFA is used as a basis of lexical analyser (LA), with 57 final states. Each state is associated with a token class, with non-final states having "null" and final states having the tokens or keywords of the CFG.
- Input to the Lexical Analyser is read each line at a time. The line is fed into the DFA.DFA generates all the tokens for the given line, corresponding to the token classes of the states.
- A dictionary is maintained to store the lexeme and its token class, which is written back to the output file.

Note: DFA also checks for lexical errors. In case there is any error class ("null") detected, the user is given a prompt, before the output file is generated.

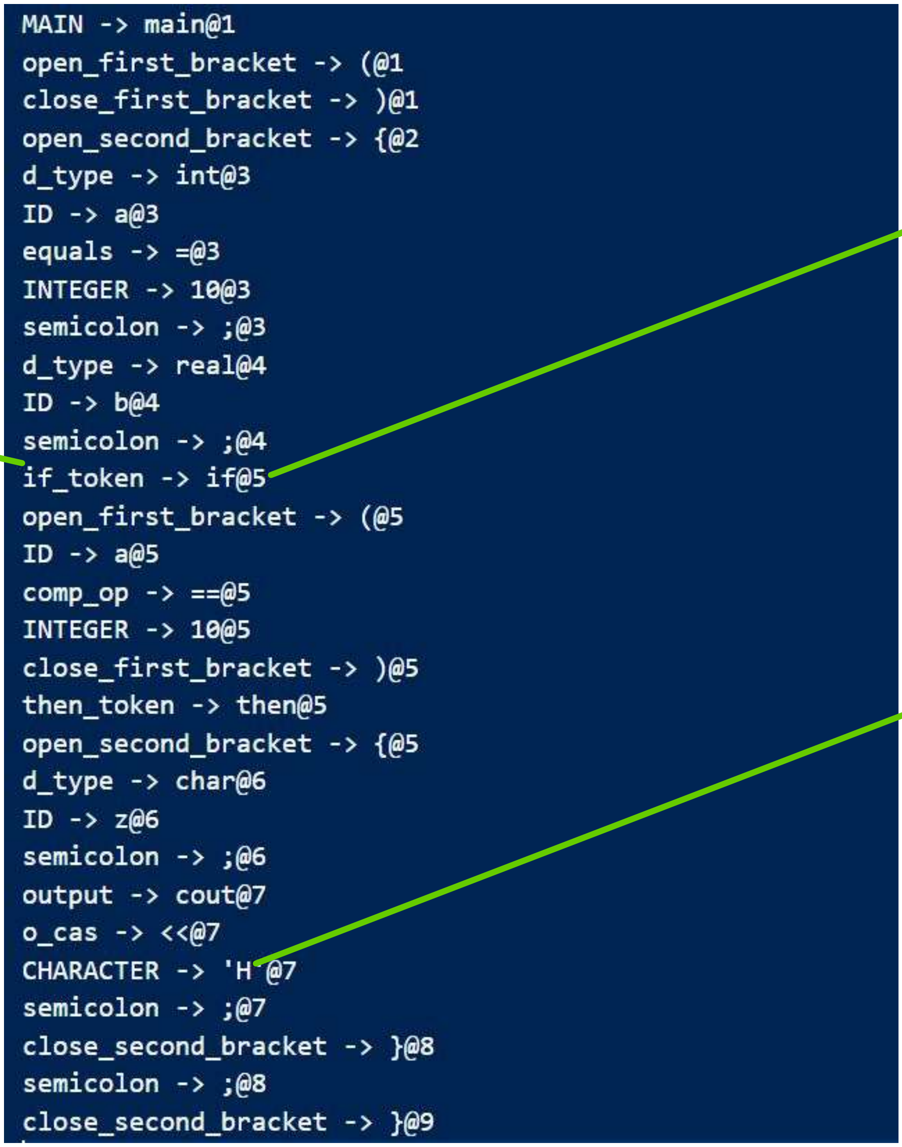
Lexical Analysis (Sample DFA and its implementation)





```
// manually adding all the transitions in dfa for 61 states
dfa->addNextStateForAState(0, 1, '+');
dfa->addNextStateForAState(0, 2, '-');
dfa->addNextStateForAState(0, 3, '/');
dfa->addNextStateForAState(0, 4, '*');
dfa->addNextStateForAState(0, 5, '%');

dfa->addNextStateForAState(0, 6, '<');
dfa->addNextStateForAState(0, 9, '>');
dfa->addNextStateForAState(0, 12, '=');
dfa->addNextStateForAState(0, 14, '!');
dfa->addNextStateForAState(0, 16, '(');
dfa->addNextStateForAState(0, 17, ')');
dfa->addNextStateForAState(0, 18, '{');
dfa->addNextStateForAState(0, 19, '}');

dfa->addNextStateForAState(0, 20, 'a');
dfa->addNextStateForAState(0, 23, 'c');
dfa->addNextStateForAState(0, 32, 'e');
dfa->addNextStateForAState(0, 36, 'i');
dfa->addNextStateForAState(0, 40, 'm');
dfa->addNextStateForAState(0, 44, 't');
dfa->addNextStateForAState(0, 48, '#');
dfa->addNextStateForAState(0, 52, '\'');
dfa->addNextStateForAState(0, 60, ';');
dfa->addNextStateForAState(0, 56, 'r');
dfa->addNextStateForAState(56, 57, 'e');
dfa->addNextStateForAState(57, 58, 'a');
dfa->addNextStateForAState(58, 59, 'l');


dfa->addNextStateForAState(6, 7, '<');
dfa->addNextStateForAState(6, 8, '=');
dfa->addNextStateForAState(9, 10, '>');
dfa->addNextStateForAState(9, 11, '=');
dfa->addNextStateForAState(12, 13, '=');
dfa->addNextStateForAState(14, 15, '=');
dfa->addNextStateForAState(20, 21, 'n');
dfa->addNextStateForAState(21, 22, 'd');
dfa->addNextStateForAState(23, 24, 'i');
```

Lexical Analyser (Output)



```
MAIN -> main@1
open_first_bracket -> (@1
close_first_bracket -> )@1
open_second_bracket -> {@2
d_type -> int@3
ID -> a@3
equals -> =@3
INTEGER -> 10@3
semicolon -> ;@3
d_type -> real@4
ID -> b@4
semicolon -> ;@4
if_token -> if@5
open_first_bracket -> (@5
ID -> a@5
comp_op -> ==@5
INTEGER -> 10@5
close_first_bracket -> )@5
then_token -> then@5
open_second_bracket -> {@5
d_type -> char@6
ID -> z@6
semicolon -> ;@6
output -> cout@7
o_cas -> <<@7
CHARACTER -> 'H'@7
semicolon -> ;@7
close_second_bracket -> }@8
semicolon -> ;@8
close_second_bracket -> }@9
```

Token Class

Line number

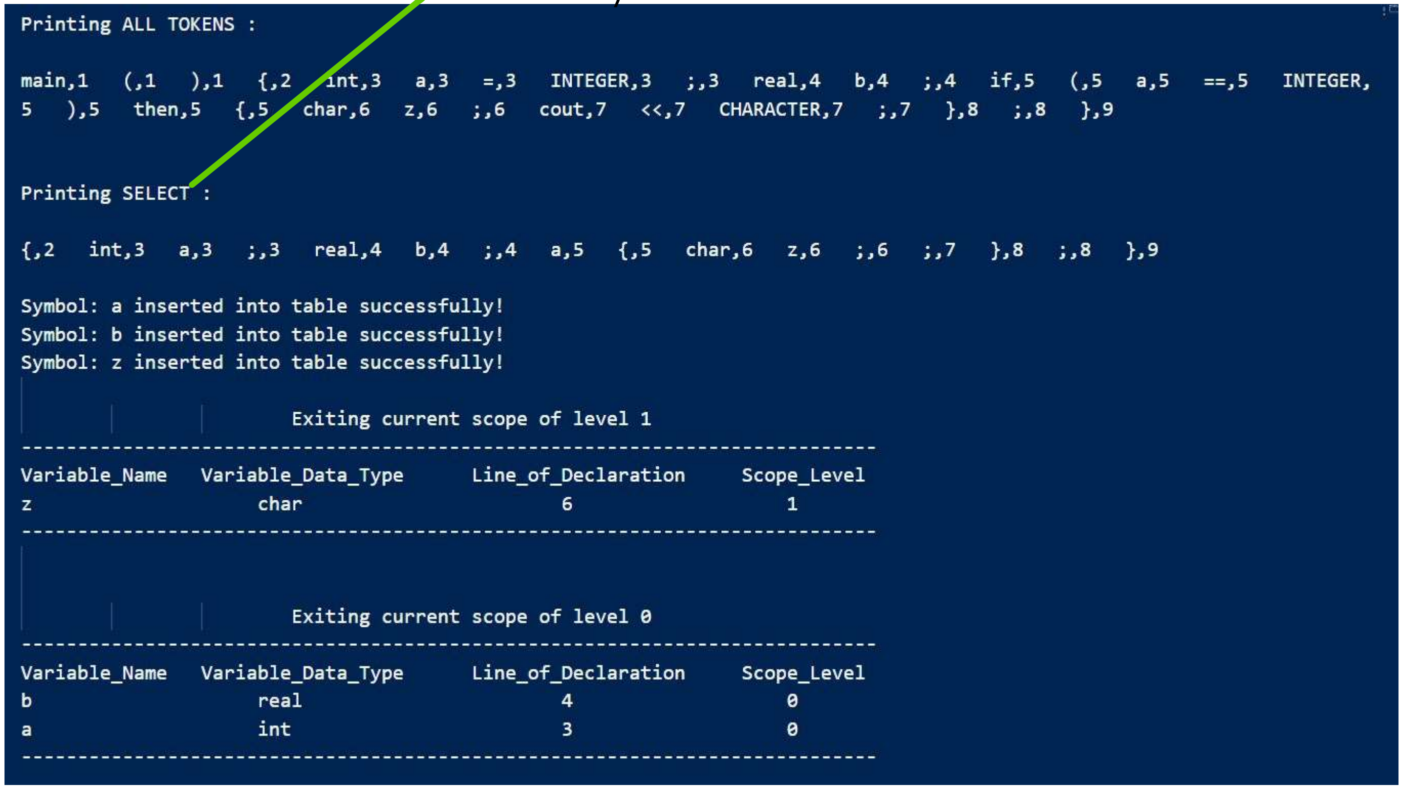
Tokens

Symbol Table

- Designed Symbol Table using Multiple Hash Tables, implemented as a dynamic list of hash tables.
- One hash table for each currently visible scope.
- Whenever a token is inserted in the symbol table, the name of the id(identifier) is the key, and the values alongside consists of attributes - data type and the line number of usage.
- During scanning, whenever it enters a new block of code, a new hash table is generated in the head of the list, and when it exits the block, the hash table in the head of the list is deleted and the symbol table of that scope is displayed in the terminal.
- If it encounters an error(usage without declaration or in wrong scope), it breaks out of loop and gives us the line in which error has occurred.

Symbol Table (Output)

SELECT prints tokens that help in building the symbol table



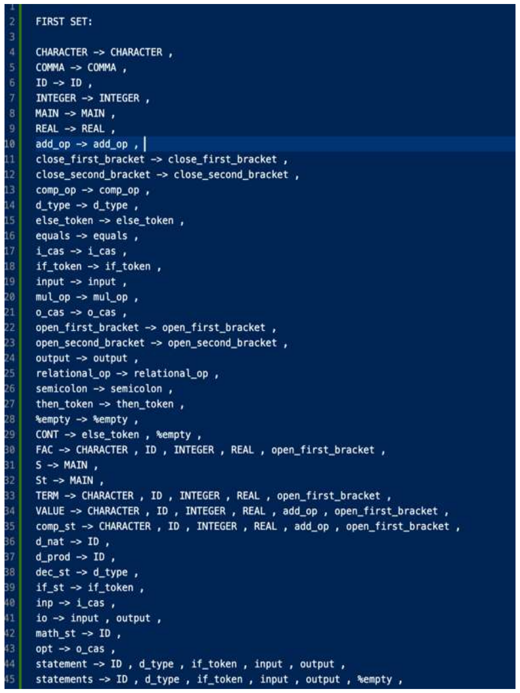
```
Printing ALL TOKENS :

main,1   (,1   ),1   {,2   int,3   a,3   =,3   INTEGER,3   ;,3   real,4   b,4   ;,4   if,5   (,5   a,5   ==,5   INTEGER,
5   ),5   then,5   {,5   char,6   z,6   ;,6   cout,7   <<,7   CHARACTER,7   ;,7   },8   ;,8   },9


Printing SELECT :

{,2   int,3   a,3   ;,3   real,4   b,4   ;,4   a,5   {,5   char,6   z,6   ;,6   ;,7   },8   ;,8   },9

Symbol: a inserted into table successfully!
Symbol: b inserted into table successfully!
Symbol: z inserted into table successfully!

                        Exiting current scope of level 1
-----------------------------------------------------------------------------
Variable_Name   Variable_Data_Type      Line_of_Declaration     Scope_Level
z                    char                       6                    1
-----------------------------------------------------------------------------


                        Exiting current scope of level 0
-----------------------------------------------------------------------------
Variable_Name   Variable_Data_Type      Line_of_Declaration     Scope_Level
b                     real                      4                    0
a                     int                       3                    0
-----------------------------------------------------------------------------
```

Bottom Up Parser

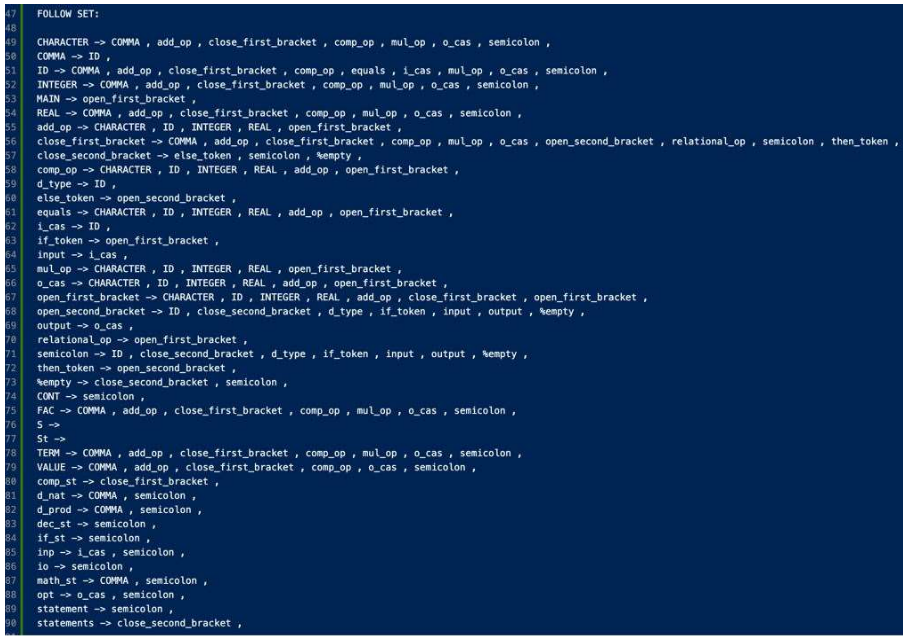
The Bottom-up Parsing is done using the SLR parsing technique.
The program receives the tokens (from lexical analyser) and the grammar as input and:

- generates the LR(0) item set with transitions
- does the calculation of first and follow sets of the grammar
- simulates the parsing process on the tokens
- prints the resulting parse tree

FIRST and FOLLOW Set

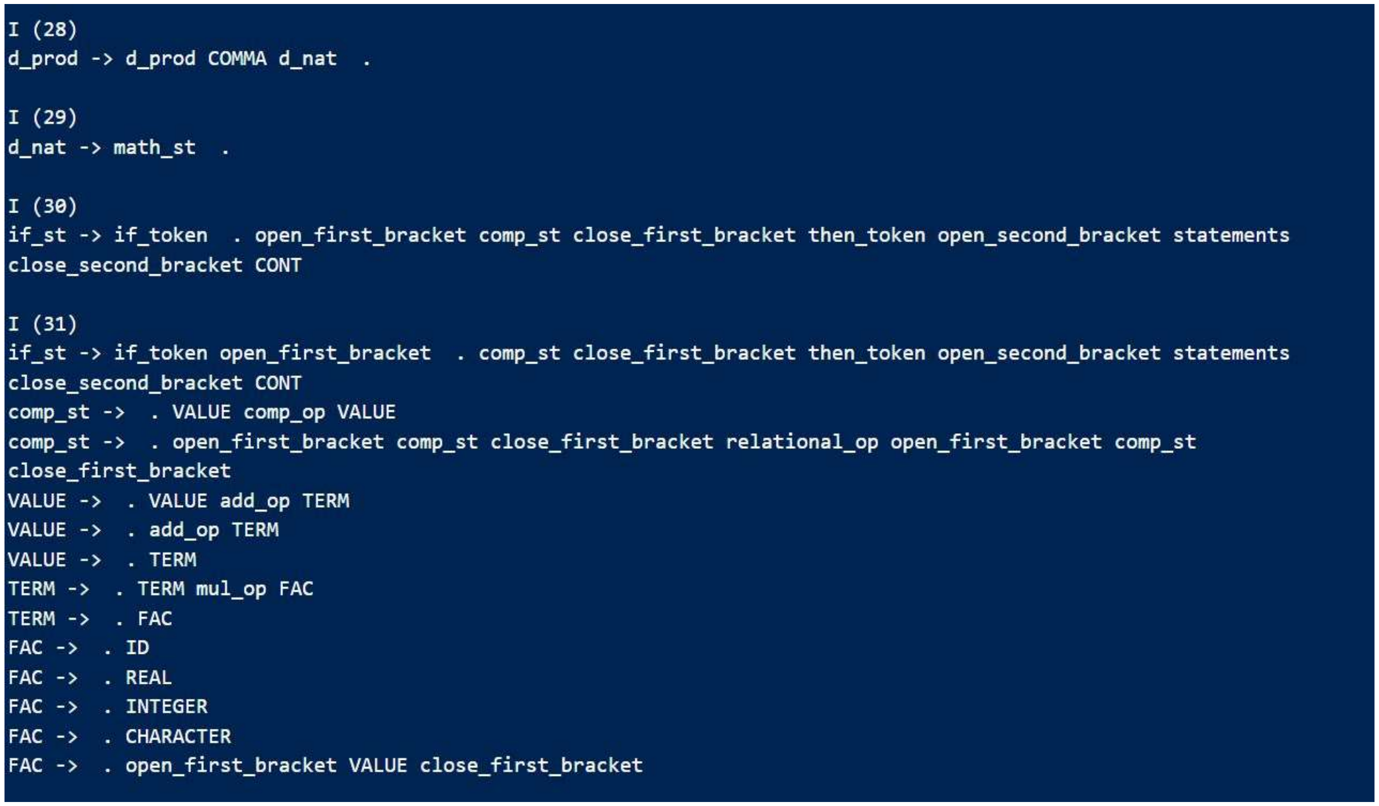


```
FIRST SET:

CHARACTER -> CHARACTER ,
COMMA -> COMMA ,
ID -> ID ,
INTEGER -> INTEGER ,
MAIN -> MAIN ,
REAL -> REAL ,
add_op -> add_op ,
close_first_bracket -> close_first_bracket ,
close_second_bracket -> close_second_bracket ,
comp_op -> comp_op ,
d_type -> d_type ,
else_token -> else_token ,
equals -> equals ,
i_cas -> i_cas ,
if_token -> if_token ,
input -> input ,
mul_op -> mul_op ,
o_cas -> o_cas ,
open_first_bracket -> open_first_bracket ,
open_second_bracket -> open_second_bracket ,
output -> output ,
relational_op -> relational_op ,
semicolon -> semicolon ,
then_token -> then_token ,
%empty -> %empty ,
CONT -> else_token , %empty ,
FAC -> CHARACTER , ID , INTEGER , REAL , open_first_bracket ,
S -> MAIN ,
St -> MAIN ,
TERM -> CHARACTER , ID , INTEGER , REAL , open_first_bracket ,
VALUE -> CHARACTER , ID , INTEGER , REAL , add_op , open_first_bracket ,
comp_st -> CHARACTER , ID , INTEGER , REAL , add_op , open_first_bracket ,
d_nat -> ID ,
d_prod -> ID ,
dec_st -> d_type ,
if_st -> if_token ,
inp -> i_cas ,
io -> input , output ,
math_st -> ID ,
opt -> o_cas ,
statement -> ID , d_type , if_token , input , output ,
statements -> ID , d_type , if_token , input , output , %empty ,
```



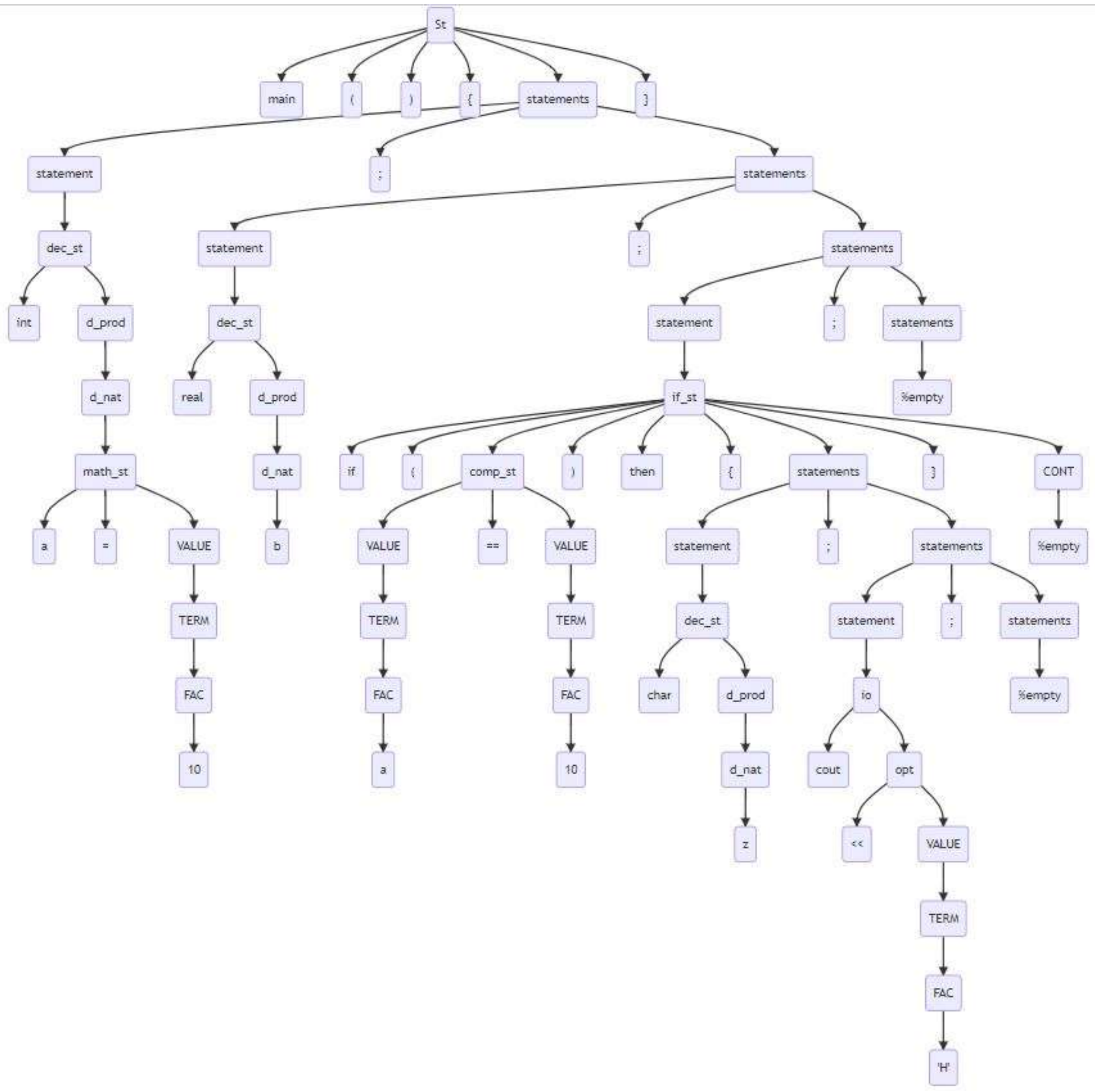
```
FOLLOW SET:

CHARACTER -> COMMA , add_op , close_first_bracket , comp_op , mul_op , o_cas , semicolon ,
COMMA -> ID ,
ID -> COMMA , add_op , close_first_bracket , comp_op , equals , i_cas , mul_op , o_cas , semicolon ,
INTEGER -> COMMA , add_op , close_first_bracket , comp_op , mul_op , o_cas , semicolon ,
MAIN -> open_first_bracket ,
REAL -> COMMA , add_op , close_first_bracket , comp_op , mul_op , o_cas , semicolon ,
add_op -> CHARACTER , ID , INTEGER , REAL , open_first_bracket ,
close_first_bracket -> COMMA , add_op , close_first_bracket , comp_op , mul_op , o_cas , open_second_bracket , relational_op , semicolon , then_token ,
close_second_bracket -> else_token , semicolon , %empty ,
comp_op -> CHARACTER , ID , INTEGER , REAL , add_op , open_first_bracket ,
d_type -> ID ,
else_token -> open_second_bracket ,
equals -> CHARACTER , ID , INTEGER , REAL , add_op , open_first_bracket ,
i_cas -> ID ,
if_token -> open_first_bracket ,
input -> i_cas ,
mul_op -> CHARACTER , ID , INTEGER , REAL , open_first_bracket ,
o_cas -> CHARACTER , ID , INTEGER , REAL , add_op , open_first_bracket ,
open_first_bracket -> CHARACTER , ID , INTEGER , REAL , add_op , close_first_bracket , open_first_bracket ,
open_second_bracket -> ID , close_second_bracket , d_type , if_token , input , output , %empty ,
output -> o_cas ,
relational_op -> open_first_bracket ,
semicolon -> ID , close_second_bracket , d_type , if_token , input , output , %empty ,
then_token -> open_second_bracket ,
%empty -> close_second_bracket , semicolon ,
CONT -> semicolon ,
FAC -> COMMA , add_op , close_first_bracket , comp_op , mul_op , o_cas , semicolon ,
S ->
St ->
TERM -> COMMA , add_op , close_first_bracket , comp_op , mul_op , o_cas , semicolon ,
VALUE -> COMMA , add_op , close_first_bracket , comp_op , o_cas , semicolon ,
comp_st -> close_first_bracket ,
d_nat -> COMMA , semicolon ,
d_prod -> COMMA , semicolon ,
dec_st -> semicolon ,
if_st -> semicolon ,
inp -> i_cas , semicolon ,
io -> semicolon ,
math_st -> COMMA , semicolon ,
opt -> o_cas , semicolon ,
statement -> semicolon ,
statements -> close_second_bracket ,
```

Bottom Up Parser (sample Item set)



```
I (28)
d_prod -> d_prod COMMA d_nat  .

I (29)
d_nat -> math_st  .

I (30)
if_st -> if_token  . open_first_bracket comp_st close_first_bracket then_token open_second_bracket statements
close_second_bracket CONT

I (31)
if_st -> if_token open_first_bracket  . comp_st close_first_bracket then_token open_second_bracket statements
close_second_bracket CONT
comp_st ->  . VALUE comp_op VALUE
comp_st ->  . open_first_bracket comp_st close_first_bracket relational_op open_first_bracket comp_st
close_first_bracket
VALUE ->  . VALUE add_op TERM
VALUE ->  . add_op TERM
VALUE ->  . TERM
TERM ->  . TERM mul_op FAC
TERM ->  . FAC
FAC ->  . ID
FAC ->  . REAL
FAC ->  . INTEGER
FAC ->  . CHARACTER
FAC ->  . open_first_bracket VALUE close_first_bracket
```

Parse Tree Example



THANK YOU