

Computer Engineering

COM241 Programming Language Concepts

Project:

Lex and Yacc

Prepared by:

Berat BAYRAM Rumeysa LAYIK

17290087 16290107

December 2018

Report Syllabus

1. Description of rules
   1. Description of variable names
      1. Rules of identifiers (IDENT)
   2. Rules of variable declarations
      1. İnteger declaration (i)
      2. String declaration (c)
   3. Rules of arithmetic operators and their precedence
      1. Addition operator (+)
      2. Subtraction operator (-)
      3. Multiplication operator (\*)
      4. Division operator (/)
      5. Modulus operator (%)
      6. Precedence of our arithmetic operators
   4. Rules of comparison operators
      1. Less than operator (<)
      2. Less than or equal than operator (<=)
      3. Greater than operator (>)
      4. Greater than or equal than operator (>=)
      5. Not equal operator (! = )
      6. Equal operator (==)
   5. Description of conditional statement
   6. Description of loop structure
2. Details of our grammar in BNF notation
   1. BNF notation of variable declarations
   2. BNF notation of arithmetic operators
   3. BNF notation of logical operators
   4. BNF notation of comparison operators
   5. BNF notation of conditional statements
   6. BNF notation of loop structure
3. Code descriptions
   1. Code description of variable names
   2. Code description of variable declarations
   3. Code description of arithmetic operators
   4. Code description of logical operators
   5. Code description of comparison operators
   6. Code description of conditional statements
   7. Code description of loop structure
4. The content of scanner.l, scanner.y files
   1. Content of scanner.l
   2. Content of scanner.y
5. Parse Trees of each example rule
   1. Parse Tree of example variable declaration
   2. Parse Tree of example arithmetic operations
   3. Parse Tree of example logical operations
   4. Parse Tree of example comparison operators
   5. Parse Tree of example conditional statements
   6. Parse Tree of example loop structure
6. Explanation of what we faced in our project
7. Parse Tree Example
8. Bibliography

**1.DESCRIPTION OF RULES**

**1.A.** Description of Variable Names

**1.A.i.** Rules of Identifiers

In our lexical code we define a regular expression such as

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

\"[a-zA-Z0-9\ ]\*\" {strcpy(yylval.c,yytext);return STRING;}

for better description and then we define a regular expression for identifiers such as

[a-zA-Z][a-zA-Z0-9\_]\* {strcpy(yylval.c,yytext);return IDENT;}

We aim an alphanumeric identifier such that user’s identifiers start with a character. It can start with either with capital character, lower-case or underscore character. After they give a character, identifiers can include any number of characters, numbers and underscores.

For example:

x *// is legal*

1y *//is not legal*

\_1y *//is legal*

\_\_x\_\_ *//is legal*

**1.B.** Rules of variable declarations

In our language we support assignment operator and we can provide to the user to ability to assign any corresponding type of value;

\= return ASSIGNMENT;

For example,

int a =11; *// is legal*

string a = ”hello”; *// is legal*

In our language we also do type-checking for each assignment so any assignment that does not match will not be supported.

For example,

int a = ”hello”; *//is not legal*

char a = 3.4; *//is not legal*

**1.B.i.** Integer declaration (num)

As we mention in previous subline, we include

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

So,

11 *//is num*

999999 *//is num*

Aaa *//is not a num*

0.99 *// is not a num*

**1.B. ii.** String declaration (string)

In our lexical part of our code we define a regular expression for character such as

\"[a-zA-Z0-9\ ]\*\" {strcpy(yylval.c,yytext);return STRING;}

So, user must define a char in the capsulated form of “[a-zA-Z]+”

“Aaa” *//is a string*

11 *//is not a string*

999999 *//is not a string*

0,999 *//is not a string*

**1.C.** Rules of arithmetic operators and their precedence

**1.C.i**. Addition operator (+)

In our lexical part of our code we define an integer and + operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"+" return PLUS;

Furthermore, for a better architecture for our language we added a semicolon for each calculating statement. Such as,

1+2; *//is legal and result is:3*

*1+2 //is not legal*

*1 ++ 2; //is not legal*

**1.C. ii.** Subtraction operator (-)

In our lexical part of our code we define an integer and - operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"-" return MINUS;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

1-2; *//is legal and result is:-1*

1-2 *//is not legal*

*1 - - 2 //is not legal*

**1.C. iii.** Multiplication operator (\*)

In our lexical part of our code we define an integer and \* operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"\*" return STAR;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

1\*2; *//is legal and result is:2*

*1\*2 // is not legal*

*1 \*\*2 // is not legal*

**1.C. iv.** Division operator (/)

In our lexical part of our code we define an integer and / operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"/" return SLASH;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

4 / 2; *//is legal and result is:2*

*1/2 // is not legal*

*1 / / 2 // is not legal*

**1.C. v.** Modulus operator (%)

In our lexical part of our code we define an integer and % operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"%" return MOD;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

1 % 2; *//is legal and result is:0*

*1 % 2 // is not legal*

*1 %% 2 // is not legal*

**1.C. vi.** Precedence of our arithmetic operations

Precedence order:

1. Multiplication (\*)-Division (/)-Modulus (%)
2. Addition (+)- Subtraction (-)

For example,

2+ 2\*3; //This statement is syntactically true and the result is: 8

**1.D.** Rules of our comparison operators

**1.D.i.** Less than operator (<)

In our lexical part of our code we define an integer and < operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"<" return ST;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

4 < 2; *//is legal and result is:0*

*1<2 // is not legal*

*1 << 2 // is not legal*

Furthermore, for a better architecture for our language we calculate for each comparison statement. So, our result is based on

0 *//if statement is false*

1 *//if statement is true*

For example,

1<2 *//This statement is syntactically true and the result is:1(true)*

9<3 *//This statement is syntactically true and the result is:0(false)*

**1.D. ii.** Less than or equal than operator (<=)

In our lexical part of our code we define an integer and < operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"<=" return SEQUAL;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

4 <= 2; *//is legal and result is:0*

*1<=2 // is not legal*

*1 <=<= 2 // is not legal*

Furthermore, for a better architecture for our language we calculate for each comparison statement. So, our result is based on

0 *//if statement is false*

1 *//if statement is true*

For example,

1<=2 *//This statement is syntactically true and the result is:1(true)*

9<=3 *//This statement is syntactically true and the result is:0(false)*

**1.D. iii.** Greater than operator (>)

In our lexical part of our code we define an integer and < operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

">" return GT;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

4 > 2; *//is legal and result is:1*

*1>2 // is not legal*

*1 >>2 // is not legal*

Furthermore, for a better architecture for our language we calculate for each comparison statement. So, our result is based on

0 *//if statement is false*

1 *//if statement is true*

For example,

1>2 *//This statement is syntactically true and the result is:0(false)*

9>3 *//This statement is syntactically true and the result is:1(true)*

**1.E. iv.** Greater than or equal than operator (>=)

In our lexical part of our code we define an integer and < operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

">=" return GEQUAL;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

4 >= 2; *//is legal and result is:1*

*1>=2 // is not legal*

*1 >= >= 2 // is not legal*

Furthermore, for a better architecture for our language we calculate for each comparison statement. So, our result is based on

0 *//if statement is false*

1 *//if statement is true*

For example,

1>=2 *//This statement is syntactically true and the result is:0(false)*

9>=3 *//This statement is syntactically true and the result is:1(true)*

**1.D. v.** Equal than operator (==)

In our lexical part of our code we define an integer and < operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"==" return EQUAL;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

4 == 2; *//is legal and result is:0*

*1==2 // is not legal*

*1 == 2; // is not legal*

Furthermore, for a better architecture for our language we calculate for each comparison statement. So, our result is based on

0 *//if statement is false*

1 *//if statement is true*

For example,

1==2 *//This statement is syntactically true and the result is:0(false)*

3==3 *//This statement is syntactically true and the result is:1(true)*

**1.D. v.** Not Equal than operator (!=)

In our lexical part of our code we define an integer and < operator such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

"!=" return NEQUAL;

Furthermore, for a better architecture for our language we add a semicolon for each calculating statement. Such as,

4 != 2; *//is legal and result is:1*

*1 != 2 // is not legal*

*1 != 2; // is not legal*

Furthermore, for a better architecture for our language we calculate for each comparison statement. So, our result is based on

0 *//if statement is false*

1 *//if statement is true*

For example,

2!=2 *//This statement is syntactically true and the result is:0(false)*

3!=1 *//This statement is syntactically true and the result is:1(true)*

**1.F.** Description of conditional statements

In our lexical part of our code we define a regular expression such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

[a-zA-Z][a-zA-Z0-9\_]\* {strcpy(yylval.c,yytext);return IDENT;}

"<" return ST;

">" return GT;

"==" return EQUAL;

"!=" return NEQUAL;

"<=" return SEQUAL;

">=" return GEQUAL;

In our language, our conditional statement’s name is *“if. User can code actions which only be executed if condition holds.*In conditional statement, user can write such a statement that will be written inside if

* Logical statement
* Conditional statement
* Arithmetic statement
* Identifier
* Num

In conditional statement, user can write such a statement that if the given statement holds(then part)

* Logical statement
* Conditional statement
* Arithmetic statement
* Identifier
* Num

**1.E.i.** If-statement

In this if statement, user can only define what will happen if the given condition holds.

For example,

If 1 { st1 }; *//This statement is syntactically true*

If 1 > 2 { st1 }; *//This statement is syntactically true*

If a1 { st1 }; *//This statement is syntactically true*

If 9/3 { st1 }; *//This statement is syntactically true*

If x\*y { st1 }; *//This statement is syntactically true*

**1.G.** Description of loop structure

**1.G.i.** While loop

As we mention in previous subline, we include

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

[a-zA-Z][a-zA-Z0-9\_]\* {strcpy(yylval.c,yytext);return IDENT;}

In our lexical part of our code we define a regular expression such as:

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

[a-zA-Z][a-zA-Z0-9\_]\* {strcpy(yylval.c,yytext);return IDENT;}

"<" return ST;

">" return GT;

"==" return EQUAL;

"!=" return NEQUAL;

"<=" return SEQUAL;

">=" return GEQUAL;

In our language, our loop structure’s name is *“while”.*User must define a conditional statement after the while in the capsulated form of “(statement)”. Then, user must define after the condition statement the capsulated form of “{statement}” which will be generated until conditional statement does not hold.

After the while conditional statement, user can write such a statement ( () part)

* Logical statement
* Conditional statement
* Arithmetic statement
* Identifier
* Num

In conditional statement, user can write such a statement that if the given statement holds ({} part)

* Logical statement
* Conditional statement
* Arithmetic statement
* Identifier
* Num

For example,

while 1 {st1}; *//This statement is syntactically true*

while 1<2 {st1}; *//This statement is syntactically true*

while a1 {st1}; *//This statement is syntactically true*

while a1\*b {st1}; *//This statement is syntactically true*

while a1+2 {st1}; *//This statement is syntactically true*

2. DETAILS OF OUR GRAMMAR IN BNF NOTATION

**2.A.** BNF notation of variable declarations

As we mention in previous subline, we include

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

\"[a-zA-Z0-9\ ]\*\" {strcpy(yylval.c,yytext);return STRING;}

[a-zA-Z][a-zA-Z0-9\_]\* {strcpy(yylval.c,yytext);return IDENT;}

In our yacc part of our code we define a grammer such as:

stmts : stmt SEMICOLON {;}

| COMMENT stmts {;}

| stmt SEMICOLON stmts {;}

;

stmt : TYPE\_INT IDENT {saveInt($2,0);}

| TYPE\_INT IDENT ASSIGNMENT exp {saveInt($2,$4);}

| TYPE\_STR IDENT {saveStr($2,"");}

| TYPE\_STR IDENT ASSIGNMENT STRING {saveStr($2,$4);}

| IDENT ASSIGNMENT STRING {modifyStr($1,$3);}

| IDENT ASSIGNMENT exp {modifyInt($1,$3);}

**2.B.** BNF notation of arithmetic operators

exp : factor {$$ = $1;}

| exp PLUS factor {$$ = $1 + $3;}

| exp MINUS factor {$$ = $1 - $3;}

;

factor : term {$$ = $1;}

| factor STAR term {$$ = $1 \* $3;}

| factor SLASH term {$$ = $1 / $3;}

| factor MOD term {$$ = $1 % $3;}

;

term : INTEGER {$$ = $1;}

| IDENT {$$ = getIntValue($1);}

;

**2.C.** BNF notation of logical operators and loops

stmt : IF condt scope {;}

| WHILE condt scope {;}

;

condt : exp GT condt

| exp ST condt

| exp EQUAL condt

| exp NEQUAL condt

| exp SEQUAL condt

| exp GEQUAL condt

| exp

;

**3.** CODE DESCRIPTIONS

**3.A.** Code description of variable names

For example:

A2 *// is legal*

1a *//is not legal*

a999 *//is legal*

aaaaa *//is legal*

**3.B.** Code description of variable declarations

int a=0; *// is legal*

string s=5; *//is illegal*

**3.C.** Code Description of arithmetic operators

5+8; *// is legal*

10\*2;  *// is legal*

8/2; *//is legal*

9-2; *//is legal*

9%2; *//is legal*

**3.D.** Code Description of comparison operator

5<8; *// is legal*

10>2;  *// is legal*

8<=2; *//is legal*

9>=2; *//is legal*

10!=10 *//is legal*

5==2  *//is legal*

**3.E.** Code Description of conditional statement

if a1 then a1; *// is legal*

*if 1 then a1; else st2; //is legal*

**3.F.** Code Description of loop structure

while 1 {} *//is legal*

*while a*  *//is illegal*

**4.**THE CONTENT OF SCANNER.L, SCANNER.Y FILES

**4.A.** Content of scanner.l

%{

#include <stdio.h>

#include <stdlib.h>

#include "y.tab.h"

#include <string.h>

%}

%%

"int" return TYPE\_INT;

"str" return TYPE\_STR;

\= return ASSIGNMENT;

\ / \ /.\* return COMMENT;

[0-9][0-9]\* {yylval.i = atoi(yytext);return INTEGER;}

\"[a-zA-Z0-9\ ]\*\" {strcpy(yylval.c,yytext);return STRING;}

"exit" return EXIT;

; return SEMICOLON;

: return COLON;

"+" return PLUS;

"-" return MINUS;

"\*" return STAR;

"/" return SLASH;

"%" return MOD;

"if" return IF;

"while" return WHILE;

"else" return ELSE;

"{" return OPEN\_SCOPE;

"}" return CLOSE\_SCOPE;

"output" return OUTPUT;

"outputc" return OUTPUTC;

"input" return INPUT;

"inputc" return INPUTC;

"<" return ST;

">" return GT;

"==" return EQUAL;

"!=" return NEQUAL;

"<=" return SEQUAL;

">=" return GEQUAL;

[a-zA-Z][a-zA-Z0-9\_]\* {strcpy(yylval.c,yytext);return IDENT;}

[ \t] ;

\n yylineno++;

. printf("ERROR: Invalid token at line %d\n",yylineno);

%%

int yywrap(void)

{

return 1;

}

**5.**Explanation of what we faced in our project

In this project we search on internet-based web site for increasing our knowledge about lex and yacc. Indeed, it was hard. We check multiple example about lex and yacc code that can be found in the internet. We read every lex and yacc pdf that included in our lecture’s website.

We try to generate and write on our version of them and try to examinate the best. After reading our pdfs and writing the codes that included, we try to understand what we were in charge of first we participate our requirement in a small section in order we did

* Regular expressions of variable names
* BNF of Arithmetic operations
* BNF of Logical operators
* BNF of Comparison operator
* BNF of Conditional statement
* BNF of Loop structure
* BNF of Variable declarations

For each step we try an average input and look for the bugs. Try to evaluate our architecture of language is efficient and bug-free. As an example, after if statement we see that our arithmetic operations are not be calculated. So, as a conclusion, we change our architecture and syntax of arithmetic operations. Before the change there was not a semicolon after each arithmetic calculation statement. We were indeed happy solve this bug.

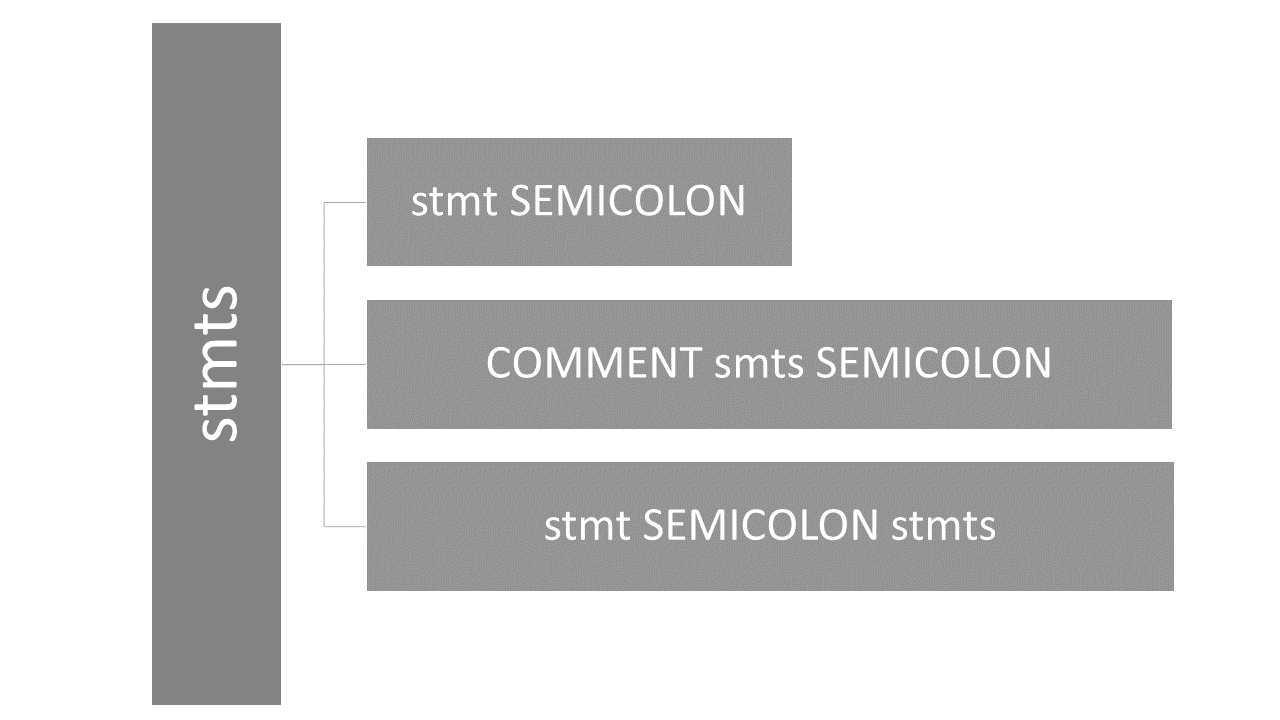
Moreover, when we describing our definition at first we could not understand where we should write this definition. We tried to write our definition in lexical part but it did not execute our given input. We could not solve this problem for a while after searching on the internet we understand what we did wrong. Definition should not be in lexical part, it must be in yacc part. After declared in our definition in yacc part we solve this problem.

In addition, in every code that we found in internet when we tried to run our computers, it always gave multiple definitions error or –lDENT return 1, may could not found, yylex could not run… We try hard to understand structure of lex and yacc what was the problem.

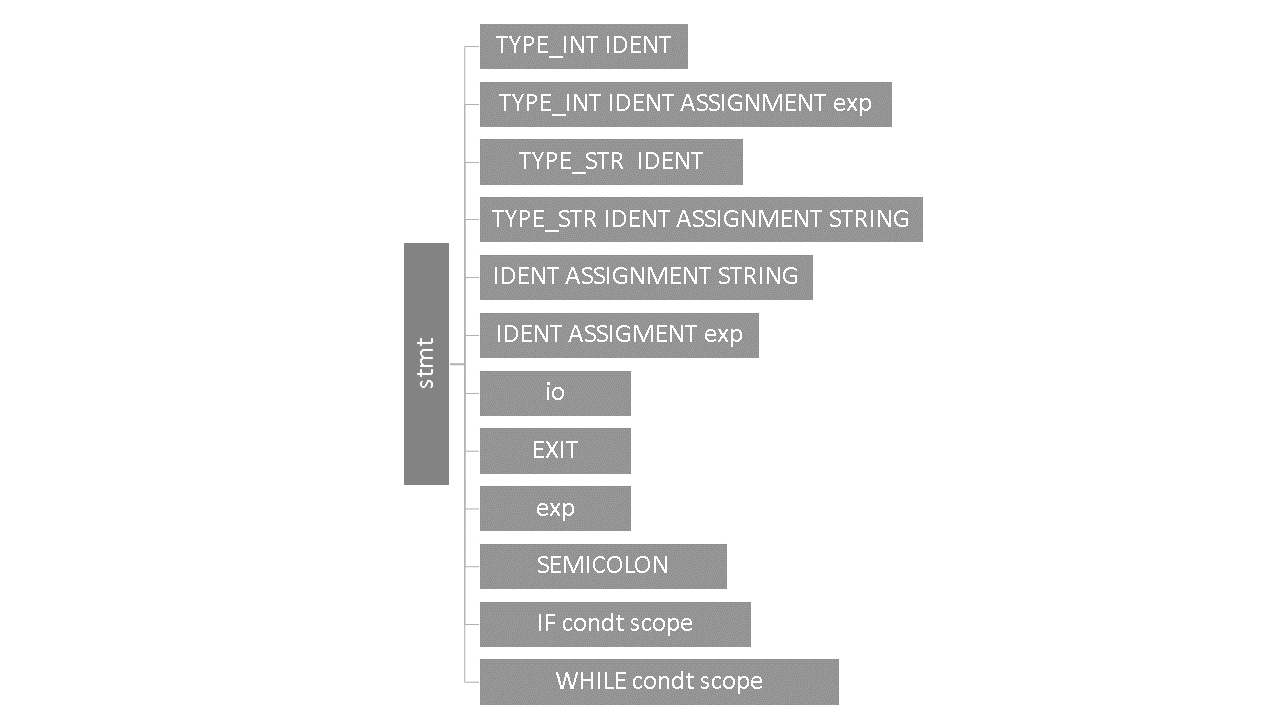
Lastly, when we finish all our project we realized that we did not do anything the type-checking .At first, we try “string a=3; “ input and it was working. So, we decide to change our architecture of our language we change all of the old BNF notation of description and for each case (int ,string) we change the way how the parser parse in the parse tree. We separate each case (int,string) to parse in different way also, we realize that as much as we check the created type we must checked type of assigned value for that we describe what is int or string in our lexical part and check the assigned value in BNF notation for each case. And it was the last major problem that we faced.

We worked hard in this project to increase our knowledge about lex and yacc and how to code it and also understanding errors in compiler and how to solve that errors.

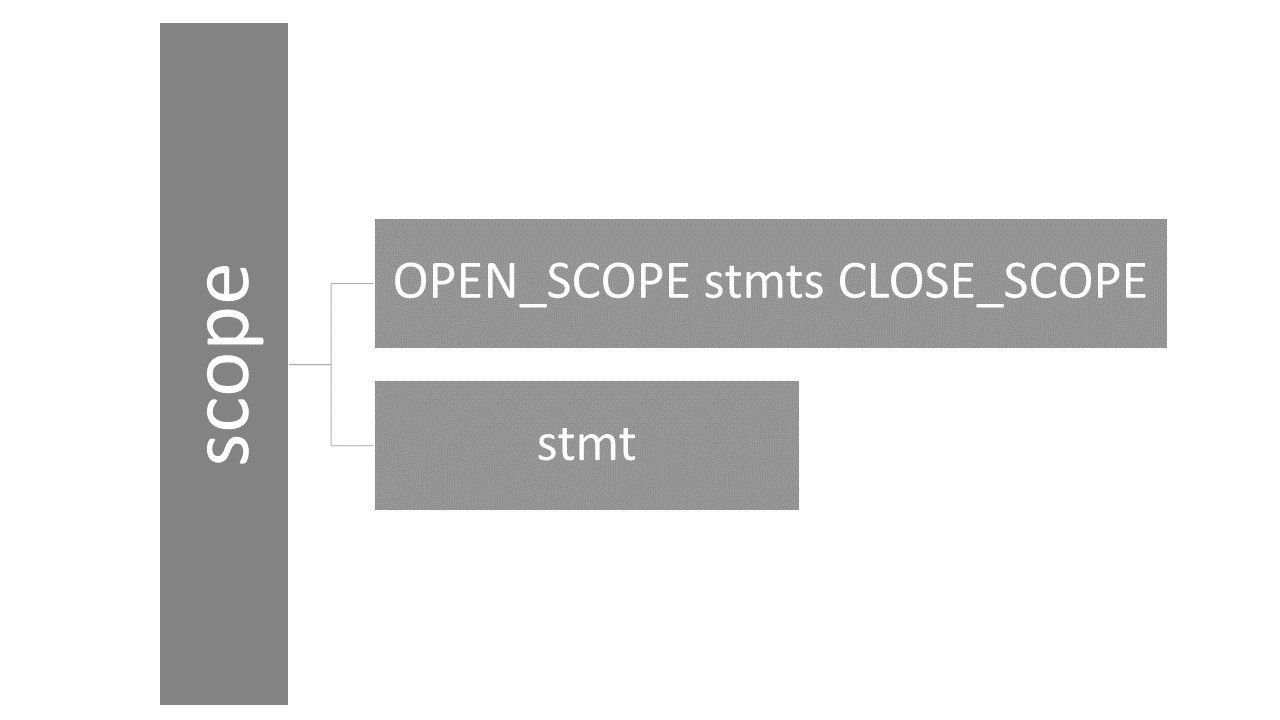
**6.** PARSETREES OF EACH EXAMPLE RULE

**6.A.** ParseTree of Program Root

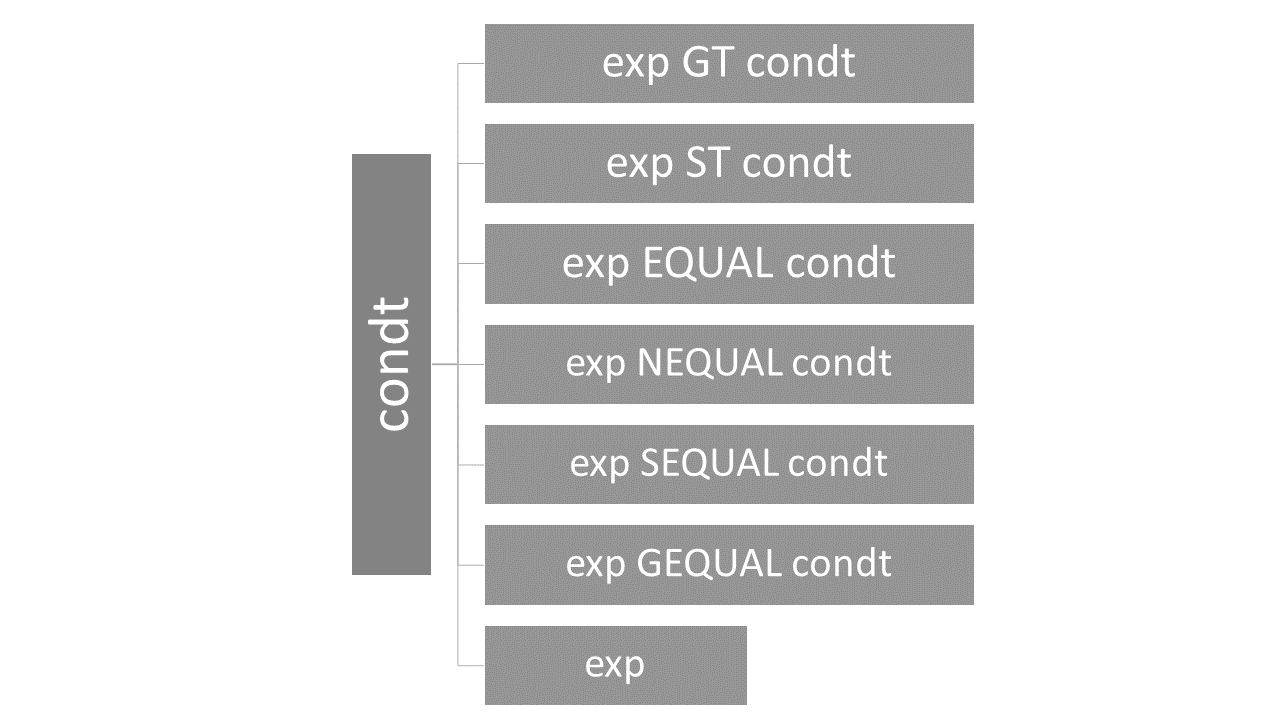
**6.B.** BNF Tree of statements



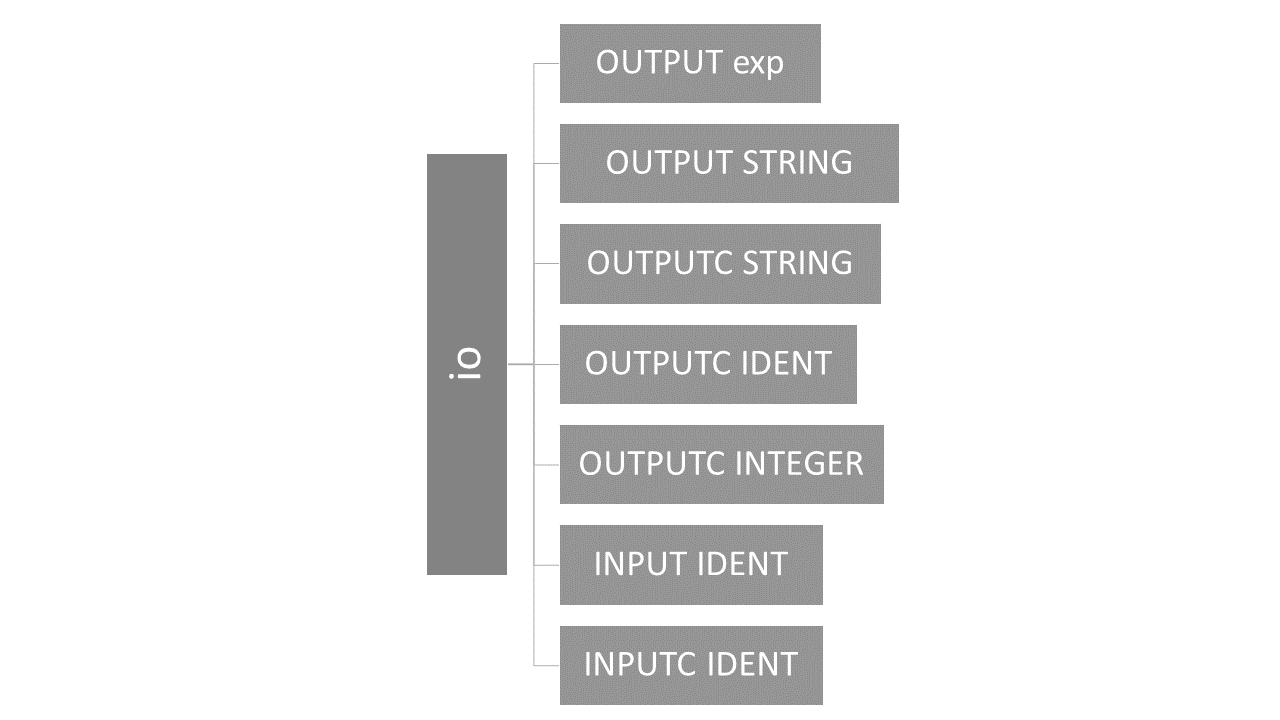
**6.C.** BNFTree of scope



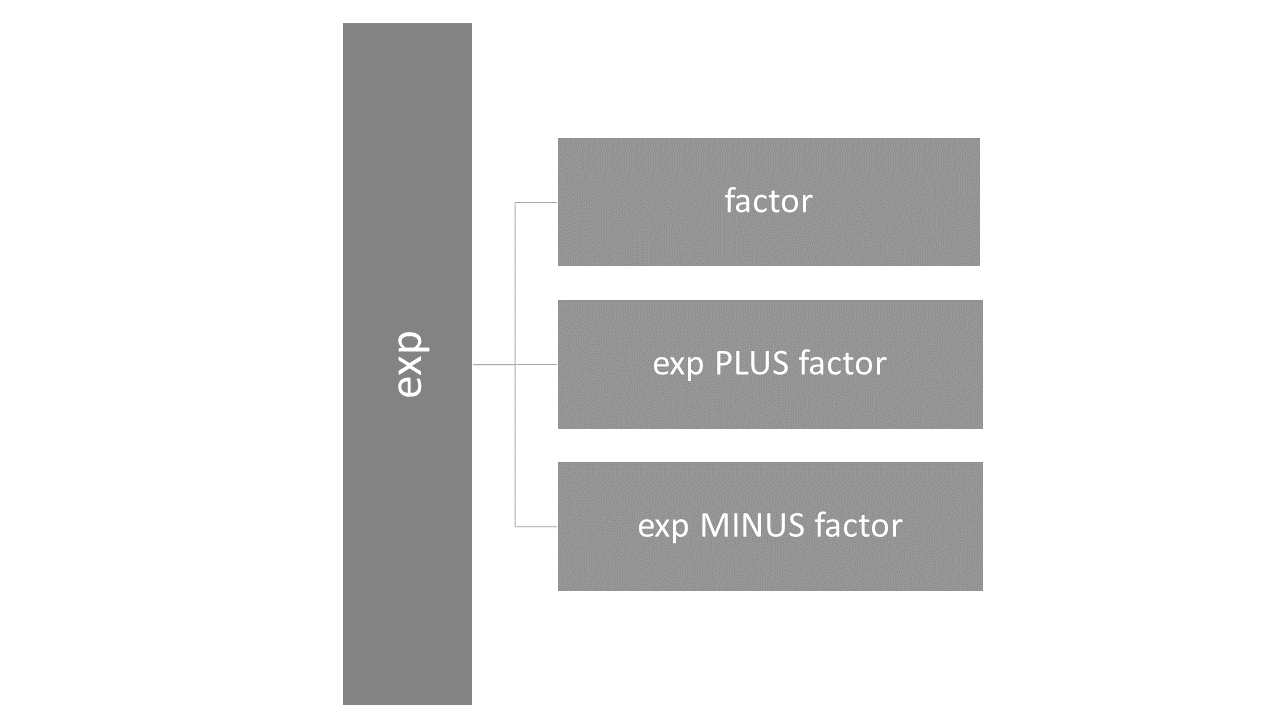
**6.D.** BNFTree conditionals



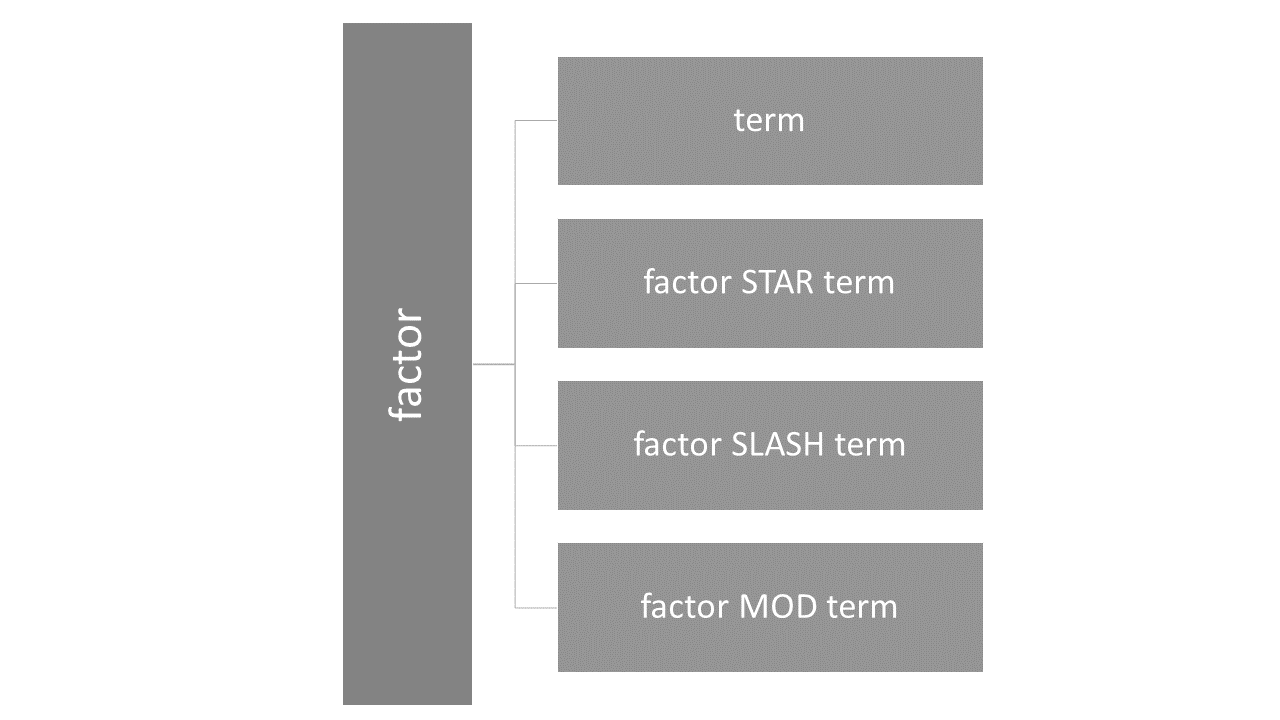
**6.E.** BNF Tree of io (input/output)



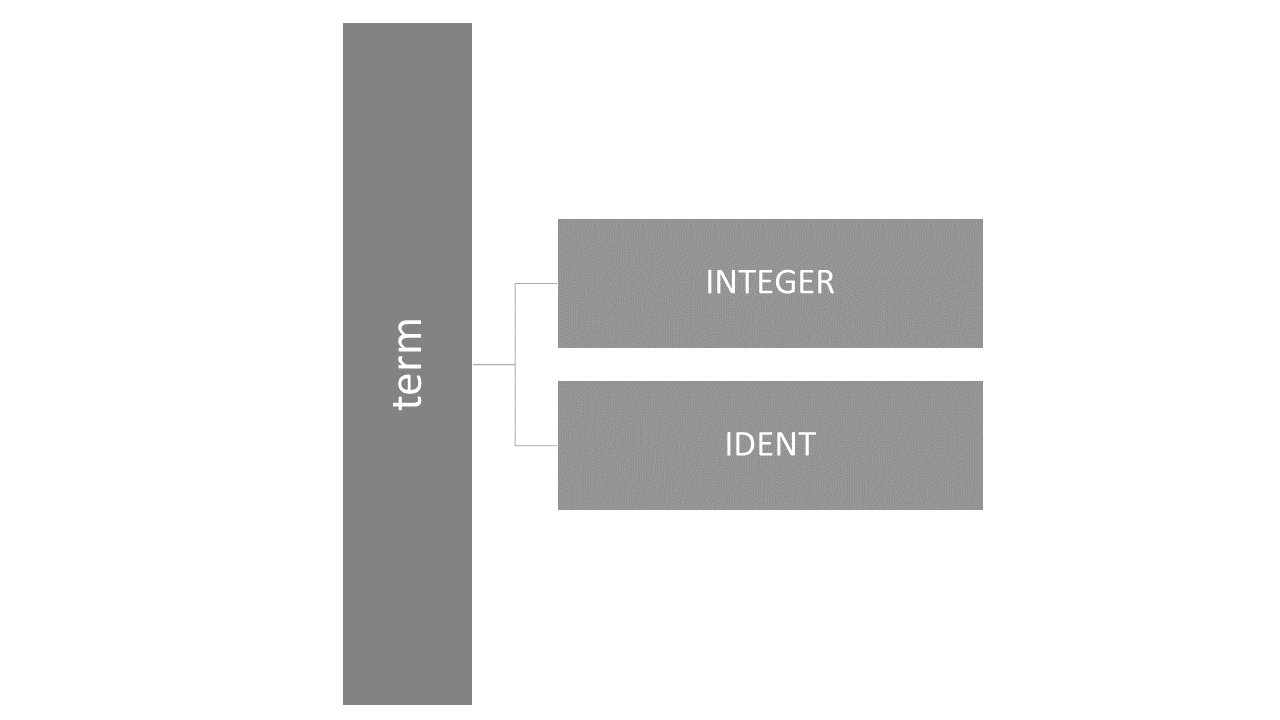
**6.F. BNF Tree of exp (expression)**

****

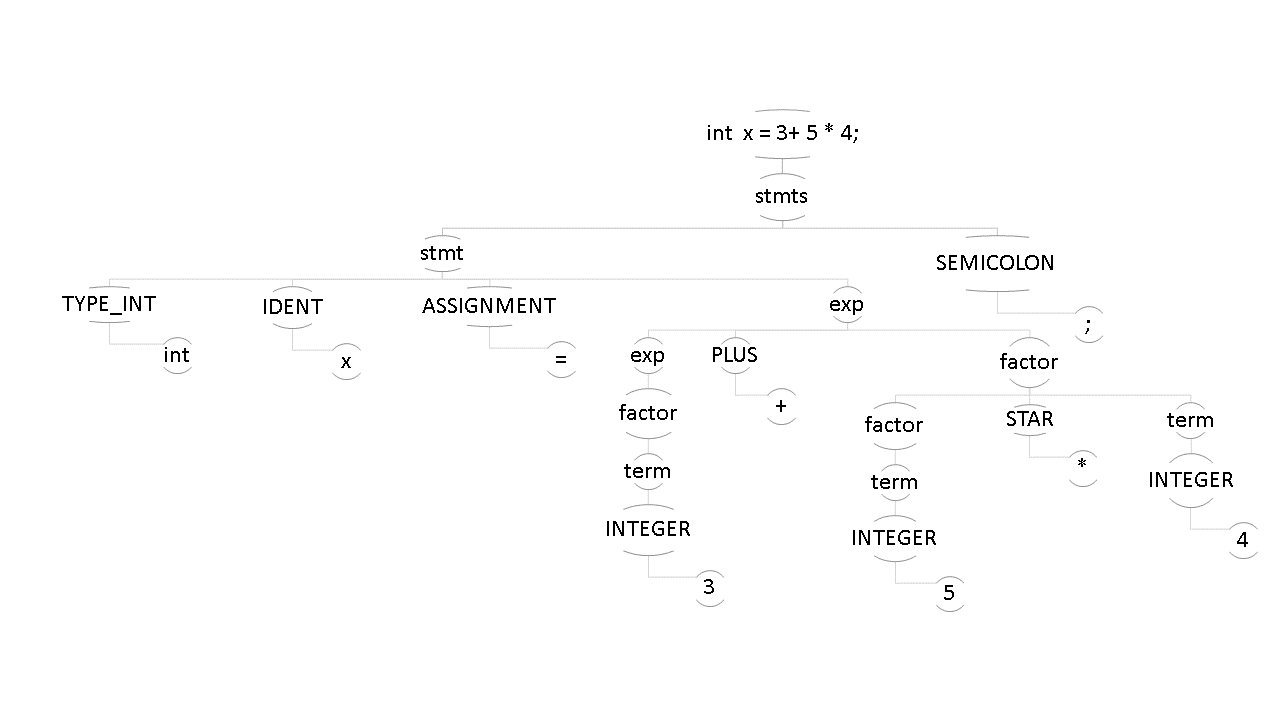
**6.G.BNF Tree of factor**

****

**6.H. BNF Tree of term**

****

**7.**PARSE TREE EXAMPLE

****

**8.**BIBLIOGRAPHY

Website that we used in this project:

<https://stackoverflow.com/questions/15641256/bison-conflicting-type-for-yyerror>

<https://stackoverflow.com/questions/15375960/runtime-syntax-error-from-lex-and-yacc>

<https://www.youtube.com/watch?v=BocfpYSGYNE>

<https://www.youtube.com/watch?v=yTXCPGAD3SQ>

<https://www.youtube.com/watch?v=54bo1qaHAfk&t=732s>

<http://en.wikipedia.org/wiki/Small-C>

<http://flex.sourceforge.net/>

<http://dinosaur.compilertools.net/yacc/>

<https://www.regextester.com/>

<https://github.com/aycavanli/lex_yacc>