

Computer Engineering

COM241 Concept of Programming Language

Project of Lex and Yacc



Prepared by

Ayça Nur VANLI Özge ÇETİN

17290128 16290086

December 2018

Report Syllabus

1. Description of the rules which we designed
   1. Description of variable names
      1. Rules of identifiers (id)
   2. Rules of variable declarations
      1. İnteger declaration (num)
      2. Float declaration (float)
      3. Character declaration (char)
      4. String declaration (string)
   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 logical operators and their precedence
      1. AND operator (&)
      2. OR operator (|)
      3. XOR Operator (x)
      4. Precedence of our logical operators
   5. 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 (==)
   6. Description of conditional statement
      1. If-then statement
      2. If- then-else statement
   7. Description of loop structure
      1. While loop
2. Details of our grammar in EBNF notation
   1. EBNF notation of variable declarations
   2. EBNF notation of arithmetic operators
   3. EBNF notation of logical operators
   4. EBNF notation of comparison operators
   5. EBNF notation of conditional statements
   6. EBNF 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 mpl.l, mpl.y files
   1. Content of mpl.l
   2. Content of mpl.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. Bibliography

**1**.Description of the rules which we designed

**1.A.** Description of variable names

**1.A.i.** Rules of identifiers

In our lexical code we define a regular expression such as

alpha [A-Za-z]

digit [0-9]

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

{alpha}({alpha}|{digit})\* return ID;

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

For example:

A2 *// is legal*

1a *//is not legal*

a999 *//is legal*

aaaaa *//is legal*

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

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

[=,{}();] {return \*yytext;}

For example,

int a=11; *//is legal*

char a=’a’; *//is legal*

string a=”hello”; *//is legal*

float a=3.5; *//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*

string a=2; *//is not legal*

float a=’a’; *//is not legal*

But in our language support assignments that can not be the same but can be converted to each other. This is shown in the table.

|  |  |
| --- | --- |
| Int | Float, Char |
| Float | Float, int |
| Char | Char, int |
| String | string |

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

As we mention in previous subline, we include

digit [0-9]

Therefore, we used a regular expression for num is

{digit}+ { yylval = atoi (yytext); return NUM;}

So,

11 *//is num*

999999 *//is num*

Aaa *//is not a num*

0.99 *// is not a num*

**1.B. ii.** Float declaration (float)

In our lexical part of our code we define a regular expression for floating-point number such as

FLOAT [0-9]+\.[0-9]+

So,

0.99 *// is a float*

Aaa *//is not a float*

11 *//is not a float*

999999 *//is not a float*

0,999 *//is not a float*

**1.B. iii.** Character declaration (char)

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

CHAR '[a-zA-Z]'

So, user must define a char in the capsulated form of ‘[a-zA-Z]’

‘a’ *//is a char*

11 *//is not a char*

999999 *//is not a char*

0,999 *//is not a char*

“Aaa” *//is not a char*

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

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

STRING \"[a-zA-Z]+\"

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 additional calculator, we only calculate sum of numbers.

As we mention in previous subline, we include

digit [0-9]

These generate operands in addition.

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

{digit}+ {yylval=atoi(yytext);return NUM;}

[=,{}();] {return \*yytext;}

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:3*

3+a;  *//is not legal*

a+a; *//is not legal*

3+2 *//is not legal*

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

In our additional calculator, we only calculate difference of numbers.

As we mention in previous subline, we include

digit [0-9]

These generate operands in subtraction.

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

{digit}+ {yylval=atoi(yytext);return NUM;}

[=,{}();] {return \*yytext;}

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*

3-a;  *//is not legal*

a-a; *//is not legal*

3-2 *//is not legal*

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

In our additional calculator, we only calculate multiplication of numbers.

As we mention in previous subline, we include

digit [0-9]

These generate operands in multiplication.

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

{digit}+ {yylval=atoi(yytext);return NUM;}

[=,{}();] {return \*yytext;}

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*

3\*a;  *//is not legal*

a\*a; *//is not legal*

3\*2 *//is not legal*

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

In our additional calculator, we only calculate division of numbers.

As we mention in previous subline, we include

digit [0-9]

These generate operands in division.

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

{digit}+ {yylval=atoi(yytext);return NUM;}

[=,{}();] {return \*yytext;}

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:0,5*

3/a;  *//is not legal*

a/a; *//is not legal*

3/2 *//is not legal*

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

In our additional calculator, we only calculate remainder of numbers.

As we mention in previous subline, we include

digit [0-9]

These generate operands in modulation.

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

{digit}+ {yylval=atoi(yytext);return NUM;}

[=,{}();] {return \*yytext;}

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*

3%a;  *//is not legal*

a%a; *//is not legal*

3%2 *//is not legal*

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

Precedence order:

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

For example,

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

**1.D.** Rules of logical operators and their precedence

**1.D.i.** AND operator (&)

In our logical operations, we only operate of number.

In our operation, we assuming any number that nonzero is 1 so its Boolean expression is true.

For example,

0 *// equals to false*

1 *//equals to true*

9999 *//equals to true*

As we mention in previous subline, we include

digit [0-9]

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

{digit}+ {yylval=atoi(yytext); return NUM;}

"&"|"|"|"x" {return \*yytext;}

These generate logical operands.

For example,

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

0&0 *//This statement is syntactically true and the result is:0(false)*

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

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

**1.D. ii.** OR operator (|)

In our logical operations, we only operate of number.

In our operation, we assuming any number that nonzero is 1 so its Boolean expression is true.

For example,

0 *// equals to false*

1 *//equals to true*

9999 *//equals to true*

As we mention in previous subline, we include

digit [0-9]

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

{digit}+ {yylval=atoi(yytext); return NUM;}

"&"|"|"|"x" {return \*yytext;}

These generate logical operands.

For example,

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

0|0 *//This statement is syntactically true and the result is:0(false)*

0|9 *//This statement is syntactically true and the result is:1(true)*

3|0 *//This statement is syntactically true and the result is:1(true)*

**1.D. iii.** XOR operator (x)

In our logical operations, we only operate of number.

In our operation, we assuming any number that nonzero is 1 so its Boolean expression is true.

For example,

0 *// equals to false*

1 *//equals to true*

9999 *//equals to true*

As we mention in previous subline, we include

digit [0-9]

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

{digit}+ {yylval=atoi(yytext); return NUM;}

"&"|"|"|"x" {return \*yytext;}

These generate logical operands.

For example,

1x2 *//This statement is syntactically true and the result is:0(false)*

0x0 *//This statement is syntactically true and the result is:0(false)*

0x9 *//This statement is syntactically true and the result is:1(true)*

3x0 *//This statement is syntactically true and the result is:1(true)*

**1.D. iv.** Precedence of our logical operators

Precedence order:

1.AND operator (&)

2.OR operator (|)-XOR operator (x)

For example,

1|0&0 *//This statement is syntactically true and the result is:1(true)*

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

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

In our logical operations, we only operate of number.

As we mention in previous subline, we include

digit [0-9]

These generate comparison operands.

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

{digit}+ {yylval=atoi(yytext); return NUM;}

"<" {return LT;}

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

In our logical operations, we only operate of number.

As we mention in previous subline, we include

digit [0-9]

These generate comparison operands.

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

{digit}+ {yylval=atoi(yytext); return NUM;}

"<=" return LE;

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)*

5<=5 *//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.E. iii.** Greater than operator (>)

In our logical operations, we only operate of number.

As we mention in previous subline, we include

digit [0-9]

These generate comparison operands.

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

{digit}+ {yylval=atoi(yytext); return NUM;}

">" {return GT;}

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,

6>3 *//This statement is syntactically true and the result is:1(true)*

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

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

In our logical operations, we only operate of number.

As we mention in previous subline, we include

digit [0-9]

These generate comparison operands.

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

{digit}+ {yylval=atoi(yytext); return NUM;}

">=" return GE;

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)*

5<=5 *//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.E. v.** Not equal than operator (!=)

In our logical operations, we only operate of number.

As we mention in previous subline, we include

digit [0-9]

These generate comparison operands.

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

{digit}+ {yylval=atoi(yytext); return NUM;}

"!=" return NE;

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,

6!=3 *//This statement is syntactically true and the result is:1(true)*

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

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

In our logical operations, we only operate of number.

As we mention in previous subline, we include

digit [0-9]

These generate comparison operands.

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

{digit}+ {yylval=atoi(yytext); return NUM;}

"==" return EQ;

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:1(true)*

2==5 *//This statement is syntactically true and the result is:0(false)*

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

As we mention in previous subline, we include

digit [0-9]

alpha [A-Za-z]

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

{digit}+ {yylval=atoi(yytext); return NUM;}

{alpha}({alpha}|{digit})\* return ID;

[=,{}();] {return \*yytext;}

"||" return OR;

"&&" return AND;

"<=" return LE;

"<" {return LT;}

">=" return GE;

">" {return GT;}

"==" return EQ;

"!=" return NE;

In our language, our conditional statement’s name is *“if”.*

We divide our if statement such as user can define either a condition than if is not holding and call is as *“else”* or usercan leave if as it is.

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.F.i.** If-then statement

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

For example,

If (1) then st1; *//This statement is syntactically true*

If (1>2) then st1; *//This statement is syntactically true*

If (a1) then st1; *//This statement is syntactically true*

If (1/1) then st1; *//This statement is syntactically true*

If (a/b) then st1; *//This statement is syntactically true*

If (1&&2) then st1; *//This statement is syntactically true*

**1.F. ii.** If-then statement

In this if statement, user also can what will happen if the given condition not holds.

This part in our language is called as *“else”.*

For example,

If (1) then st1; else s2; *//This statement is syntactically true*

If (1>2) then st1; else s2; *//This statement is syntactically true*

If (a1) then st1; else s2; *//This statement is syntactically true*

If (1/1) then st1; else s2; *//This statement is syntactically true*

If (a\*b) then st1; else s2; *//This statement is syntactically true*

If (1&&2) then st1; else s2; *//This statement is syntactically true*

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

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

As we mention in previous subline, we include

digit [0-9]

alpha [A-Za-z]

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

{digit}+ {yylval=atoi(yytext); return NUM;}

{alpha}({alpha}|{digit})\* return ID;

[=,{}();] {return \*yytext;}

"||" return OR;

"&&" return AND;

"<=" return LE;

"<" {return LT;}

">=" return GE;

">" {return GT;}

"==" return EQ;

"!=" return NE;

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 (1%%0) {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 EBNF notation

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

As we mention in previous subline, we include

digit [0-9]

alpha [A-Za-z]

FLOAT [0-9]+\.[0-9]+

STRING \"[a-zA-Z]+\"

CHAR '[a-zA-Z]'

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

"\n" return EOLN;

{digit}+ {yylval=atoi(yytext); return NUM;}

{alpha}({alpha}|{digit}) \* return ID;

float return F;

char return C;

string return STR;

{FLOAT} {return (FLOAT);}

{CHAR} {return (CHAR);}

{STRING} {return (STRING);}

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

S: S5 EOLN {printf("OK\n"); exit(0);}

;

S5: exp

;

exp: C varlist1 ';'

|I varlist2 ';'

|F varlist3 ';'

|STR varlist4 ';'

;

varlist1: varlist1 '=' CHAR

|varlist1 '=' ID

|varlist1 '=' NUM

|CHAR

|ID

;

varlist2: varlist2 '=' NUM

|varlist2 '=' ID

|varlist2 '=' CHAR

|varlist2 '=' FLOAT

|NUM

|ID

;

varlist3: varlist3 '=' FLOAT

|varlist3 '=' ID

|varlist3 '=' CHAR

|FLOAT

|ID

;

varlist4: varlist4 '=' STRING

|varlist4 '=' ID

|STRING

|ID

;

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

As we mention in previous subline, we include

digit [0-9]

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

"\n" return EOLN;

{digit}+ {yylval=atoi(yytext); return NUM;}

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

S: S4 EOLN {printf ("OK result is =%d\n”,$1; exit(0);}

;

S4: expr ';'

;

expr: expr '+' term {$$ =$1+$3;}

|expr '-' term {$$ =$1-$3;}

|term {$$ =$1;}

;

term: term '/' factor {$$ =$1/$3;}

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

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

|factor {$$ =$1;}

;

factor: '('expr')' {$$ =$2;}

|line {$$ =$1;}

;

line: end {$$ =$1;}

;

**2.C.** EBNF notation of logical operators

As we mention in previous subline, we include

digit [0-9]

alpha [A-Za-z]

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

"\n" return EOLN;

{digit}+ {yylval=atoi(yytext); return NUM;}

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

S: S1 EOLN {printf("OK result =%d\n",(int)$$);exit(0);}

;

S1: E1

;

E1: E\_1'|'E\_1 {$$=$1||$3;}

|E\_1'x'E\_1 {$$=($1&&(!$3))||((!$1)&&$3);}

|E\_1

;

E\_1: E\_1'&'E\_1 {$$=$1&&$3;}

|NUM {$$=$1;}

;

**2.D.** EBNF notation of comparison operators

As we mention in previous subline, we include

digit [0-9]

alpha [A-Za-z]

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

"\n" return EOLN;

{digit}+ {yylval=atoi(yytext); return NUM;}

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

S: S6 EOLN {printf("OK true/false =%d\n",(int)$$);exit(0);}

;

S6: Ex

;

Ex: Ex1 LT Ex1 {$$ =$1<$3;}

|Ex1 GT Ex1 {$$ =$1>$3;}

|Ex1 GE Ex1 {$$ =$1>=$3;}

|Ex1 LE Ex1 {$$ =$1<=$3;}

|Ex1 NE Ex1 {$$ =$1!=$3;}

|Ex1 EQ Ex1 {$$ =$1==$3;}

;

Ex1: NUM

;

**2.E.** EBNF notation of conditional statement

As we mention in previous subline, we include

digit [0-9]

alpha [A-Za-z]

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

"\n" return EOLN;

{alpha}({alpha}|{digit}) \* return ID;

{digit}+ {yylval=atoi(yytext); return NUM;}

"<=" return LE;

"<" {return LT;}

">=" return GE;

">" {return GT;}

"==" return EQ;

"!=" return NE;

"||" return OR;

"&&" return AND;

[=,{}();] {return \*yytext;}

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

S: S2 EOLN {printf("OK\n");exit(0);}

;

S2: ST

;

ST: IF '(' E2 ')' THEN ST1';' ELSE ST1';'

| IF '(' E2 ')' THEN ST1';'

;

ST1: ST

| E

;

E: ID'='E

| E'+'E

| E'-'E

| E'\*'E

| E'/'E

| E LT E

| E GT E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

| ID

| NUM

;

E2: E LT E

| E GT E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

| ID

| NUM

;

**2.F.** EBNF notation of loop structure

As we mention in previous subline, we include

digit [0-9]

alpha [A-Za-z]

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

"\n" return EOLN;

{alpha}({alpha}|{digit}) \* return ID;

{digit}+ {yylval=atoi(yytext); return NUM;}

"<=" return LE;

"<" {return LT;}

">=" return GE;

">" {return GT;}

"==" return EQ;

"!=" return NE;

"||" return OR;

"&&" return AND;

[=,{}();] {return \*yytext;}

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

S: S3 EOLN {printf("OK\n");exit(0);}

;

S3: whileloop

;

whileloop: WHILE '(' cond ')' '{' '}'

;

cond : scond

| scond logop cond

;

scond : nid

| nid relop nid

;

nid : ID

| NUM

;

logop : AND

| OR

;

relop : NE

| EQ

| LT

| LE

| GT

| GE

;

**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*

char b=’p’;  *// is legal*

float x=”hello”; *//is illegal*

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 logical operators

1&0 *// is legal*

9|0  *// is legal*

1x9 *//is legal*

1|0&0 *//is legal*

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

if(a1) then a1; *// is legal*

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

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

while(1){} *//is legal*

*while(a)*  *//is illegal*

**4.**The content of mpl.l, mpl.y files

**4.A.** Content of mpl.l

%{

#include"y.tab.h"

#include<stdio.h>

#include<stdlib.h>

%}

alpha [A-Za-z]

digit [0-9]

FLOAT [0-9]+\.[0-9]+

STRING \"[a-zA-Z]+\"

CHAR '[a-zA-Z]'

%%

"while" {return WHILE;}

if return IF;

then return THEN;

else return ELSE;

{digit}+ {yylval=atoi(yytext);return NUM;}

int return I;

float return F;

char return C;

string return STR;

{alpha}({alpha}|{digit})\* return ID;

{FLOAT} {return (FLOAT);}

{CHAR} {return (CHAR);}

{STRING} {return (STRING);}

"<=" return LE;

"<" {return LT;}

">=" return GE;

">" {return GT;}

"==" return EQ;

"!=" return NE;

"||" return OR;

"&&" return AND;

"\n" return EOLN;

"&"|"|"|"x" {return \*yytext;}

[=,{}();] {return \*yytext;}

[ \t] ;

. return \*yytext;

%%

int yywrap()

{return 1;}

**4.B.** Content of mpl.y

%{

#include<stdio.h>

#include<stdlib.h>

int i=0;

extern int yylex();

void yyerror (char \*s);

%}

%token ID NUM IF THEN LE GE EQ NE OR AND ELSE EOLN COMMA STR F C I

%token WHILE LT GT

%token FLOAT CHAR STRING

%left '&''|''x'

%left LT GT LE GE EQ NE

%left AND OR

%left '+''-'

%left '\*''/'

%right '='

%right UMINUS

%%

S: S1 EOLN {printf("OK result =%d\n",(int)$$);exit(0);}

|S2 EOLN {printf("OK\n");exit(0);}

|S3 EOLN {printf("OK\n");exit(0);}

|S4 EOLN {printf("OK result is %d\n",$1);exit(0);}

|S5 EOLN {printf("OK\n");exit(0);}

|S6 EOLN {printf("OK true/false =%d\n",(int)$$);exit(0);}

;

S1: E1

;

E1: E\_1'|'E\_1 {$$=$1||$3;}

|E\_1'x'E\_1 {$$=($1&&(!$3))||((!$1)&&$3);}

|E\_1

;

E\_1: E\_1'&'E\_1 {$$=$1&&$3;}

|NUM {$$=$1;}

;

S2: ST

;

ST: IF '(' E2 ')' THEN ST1';' ELSE ST1';'

| IF '(' E2 ')' THEN ST1';'

;

ST1: ST

| E

;

E: ID'='E

| E'+'E

| E'-'E

| E'\*'E

| E'/'E

| E LT E

| E GT E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

| ID

| NUM

;

E2 : E LT E

| E GT E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

| ID

| NUM

;

S3: whileloop

;

whileloop: WHILE '(' cond ')' '{' '}'

;

cond : scond

| scond logop cond

;

scond : nid

| nid relop nid

;

nid : ID

| NUM

;

logop : AND

| OR

;

relop : NE

| EQ

| LT

| LE

| GT

| GE

;

S4: expr ';'

;

expr: expr '+' term {$$ =$1+$3;}

|expr '-' term {$$ =$1-$3;}

|term {$$ =$1;}

;

term: term '/' factor {$$ =$1/$3;}

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

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

|factor {$$ =$1;}

;

factor: '('expr')' {$$ =$2;}

|line {$$ =$1;}

;

line: end {$$ =$1;}

;

end: NUM {$$ =$1;}

| FLOAT {$$ =$1;}

;

S5: exp

;

exp: C varlist1 ';'

|I varlist2 ';'

|F varlist3 ';'

|STR varlist4 ';'

;

varlist1: varlist1 '=' CHAR

|varlist1 '=' ID

|varlist1 '=' NUM

|CHAR

|ID

;

varlist2: varlist2 '=' NUM

|varlist2 '=' ID

|varlist2 '=' CHAR

|varlist2 '=' FLOAT

|NUM

|ID

;

varlist3:varlist3 '=' FLOAT

|varlist3 '=' ID

|varlist3 '=' CHAR

|FLOAT

|ID

;

varlist4:varlist4 '=' STRING

|varlist4 '=' ID

|STRING

|ID

;

S6: Ex

;

Ex: Ex1 LT Ex1 {$$ =$1<$3;}

|Ex1 GT Ex1 {$$ =$1>$3;}

|Ex1 GE Ex1 {$$ =$1>=$3;}

|Ex1 LE Ex1 {$$ =$1<=$3;}

|Ex1 NE Ex1 {$$ =$1!=$3;}

|Ex1 EQ Ex1 {$$ =$1==$3;}

;

Ex1: NUM

;

%%

int main()

{

yyparse();

return 0;

}

void yyerror(char \*s) {fprintf(stderr,"%s\n",s);}

**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
* EBNF of Arithmetic operations
* EBNF of Logical operators
* EBNF of Comparison operator
* EBNF of Conditional statement
* EBNF of Loop structure
* EBNF 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.

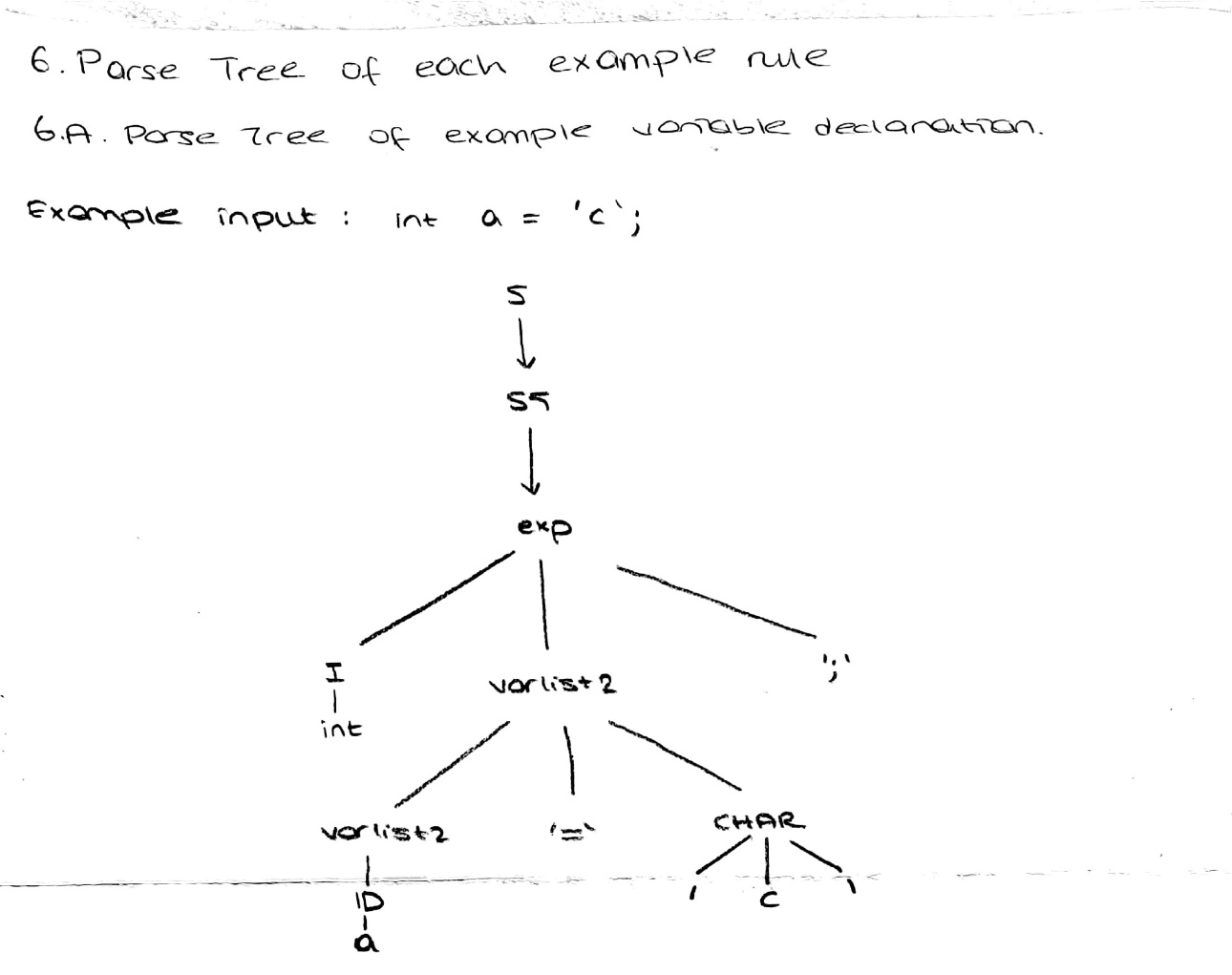
Moreover, when we describing our definition at first we could not understand where we should write this definition. We tried to write our definition in lexical part but it did not execute our given input. We could not solve this problem for a while after searching on the internet we understand what we 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 –ld return 1, may could not found, yylex could not run… We try hard to understand structure of lex and yacc what was the problem. It was very complex.

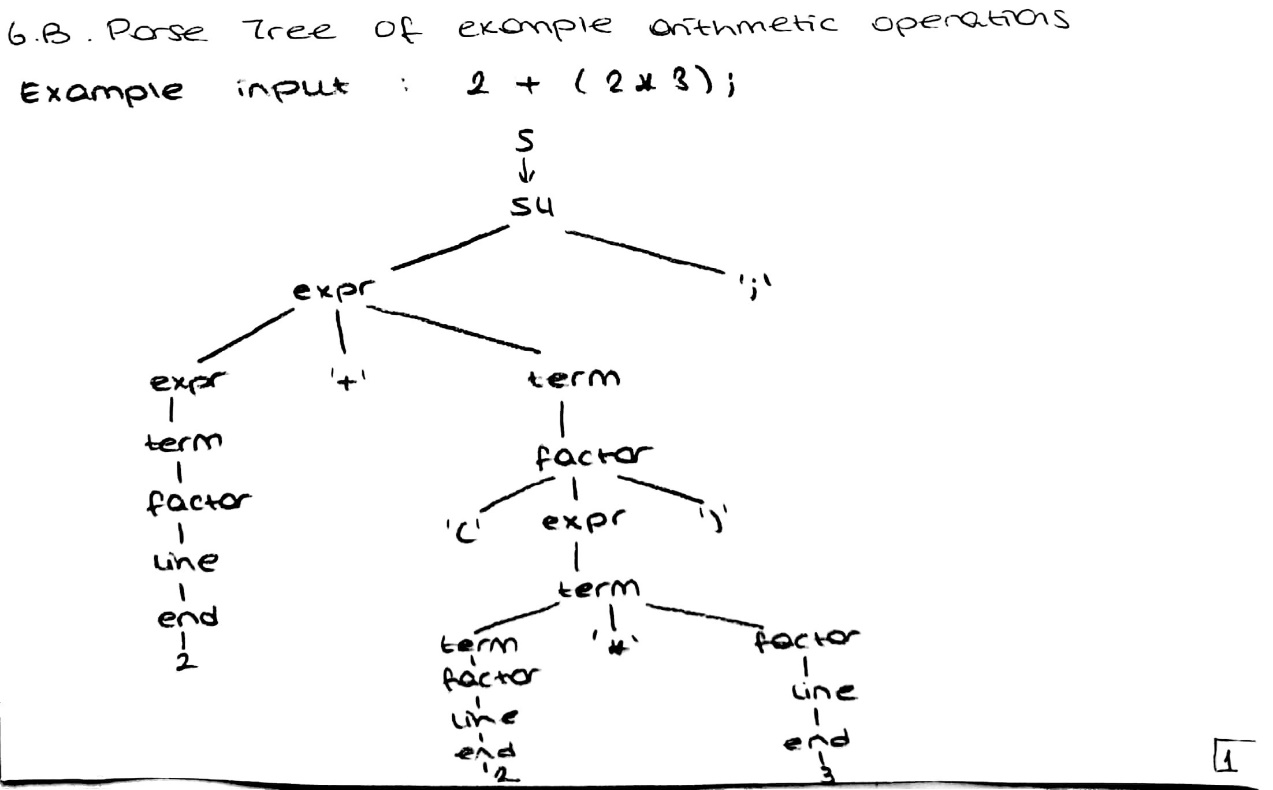
Lastly, when we finish all our project we realized that we did not do anything the type-checking .At first, we try “string a=3; “ input and it was working. So, we decide to change our architecture of our language we change all of the old EBNF notation of description and for each case (int,char,float,string)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

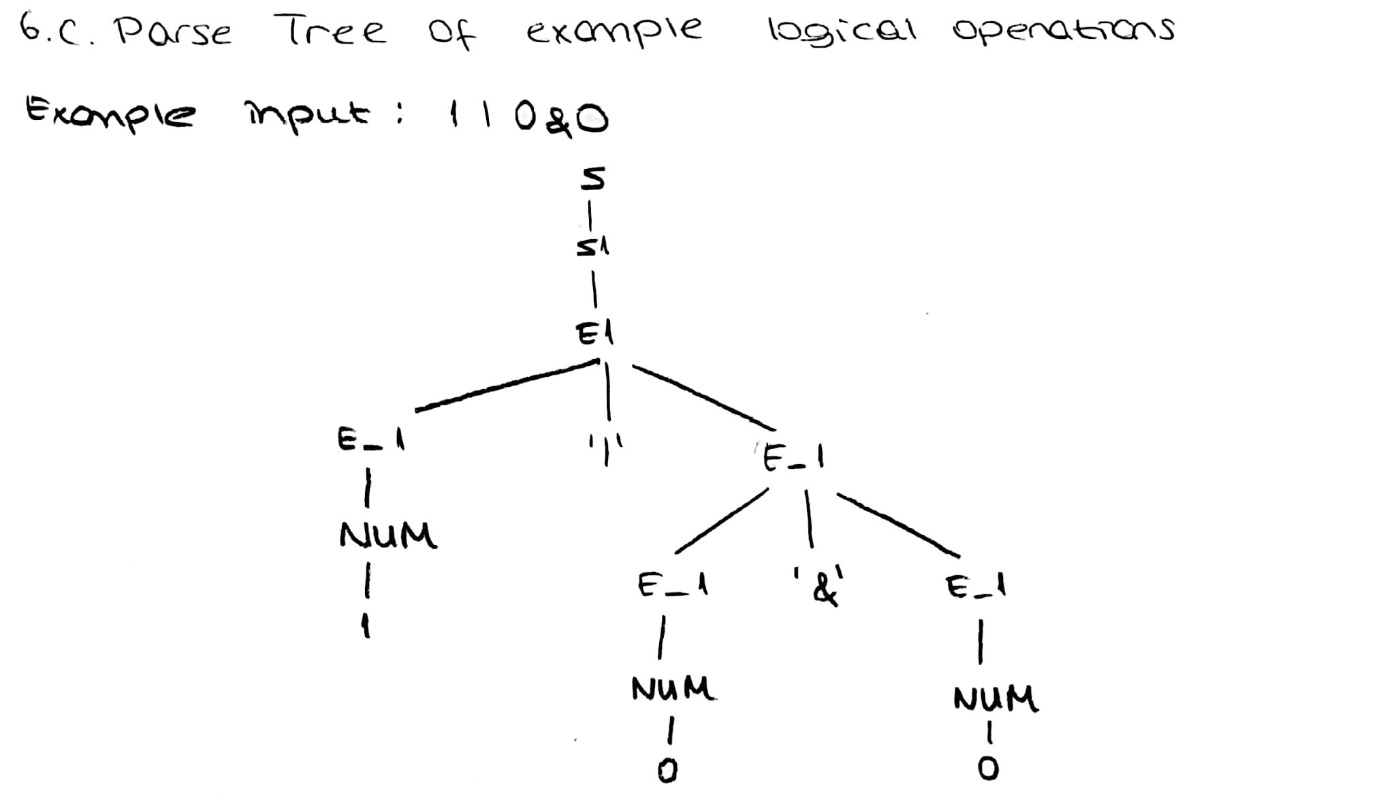
**6.** ParseTrees of each example rule

**6.A.** ParseTrees of example variable declaration****

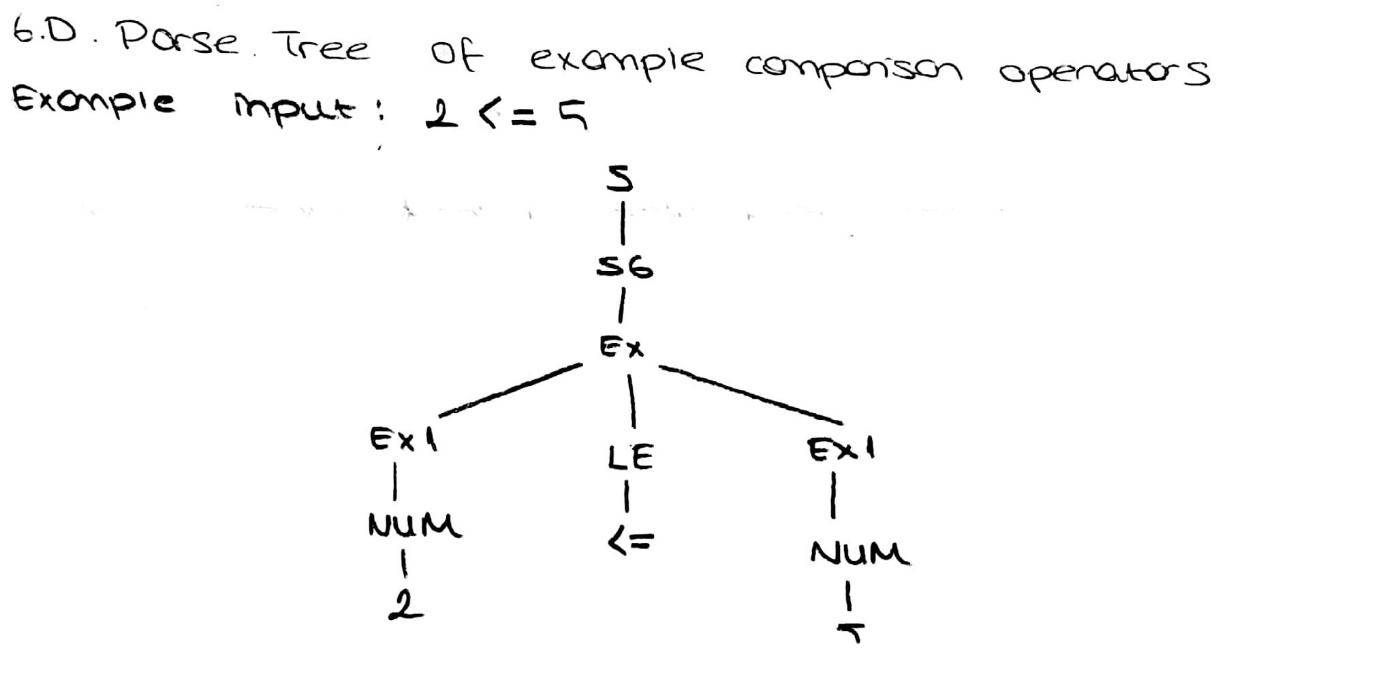
**6.B.** ParseTrees of example arithmetic operations

****

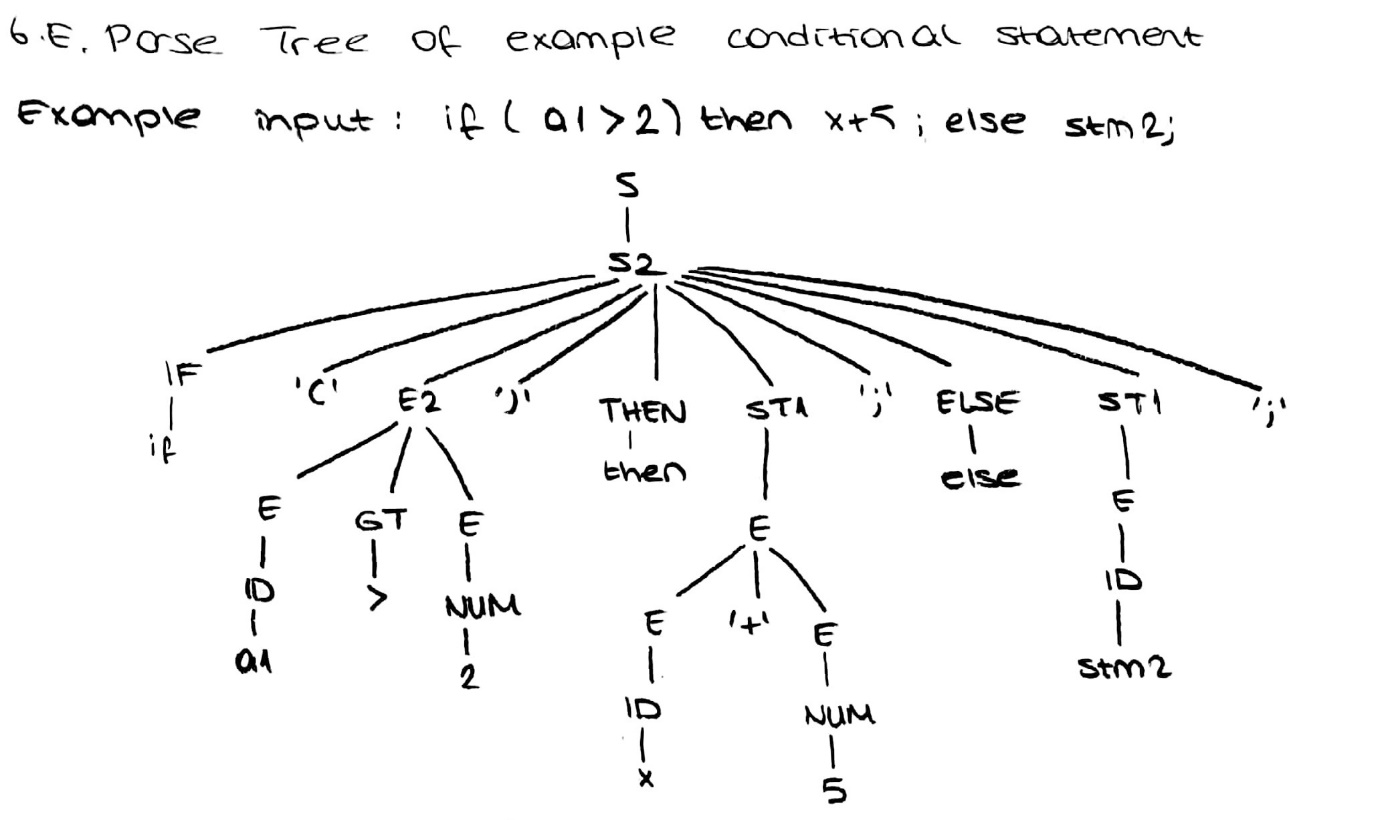
**6.C.** ParseTrees of example logical operations

****

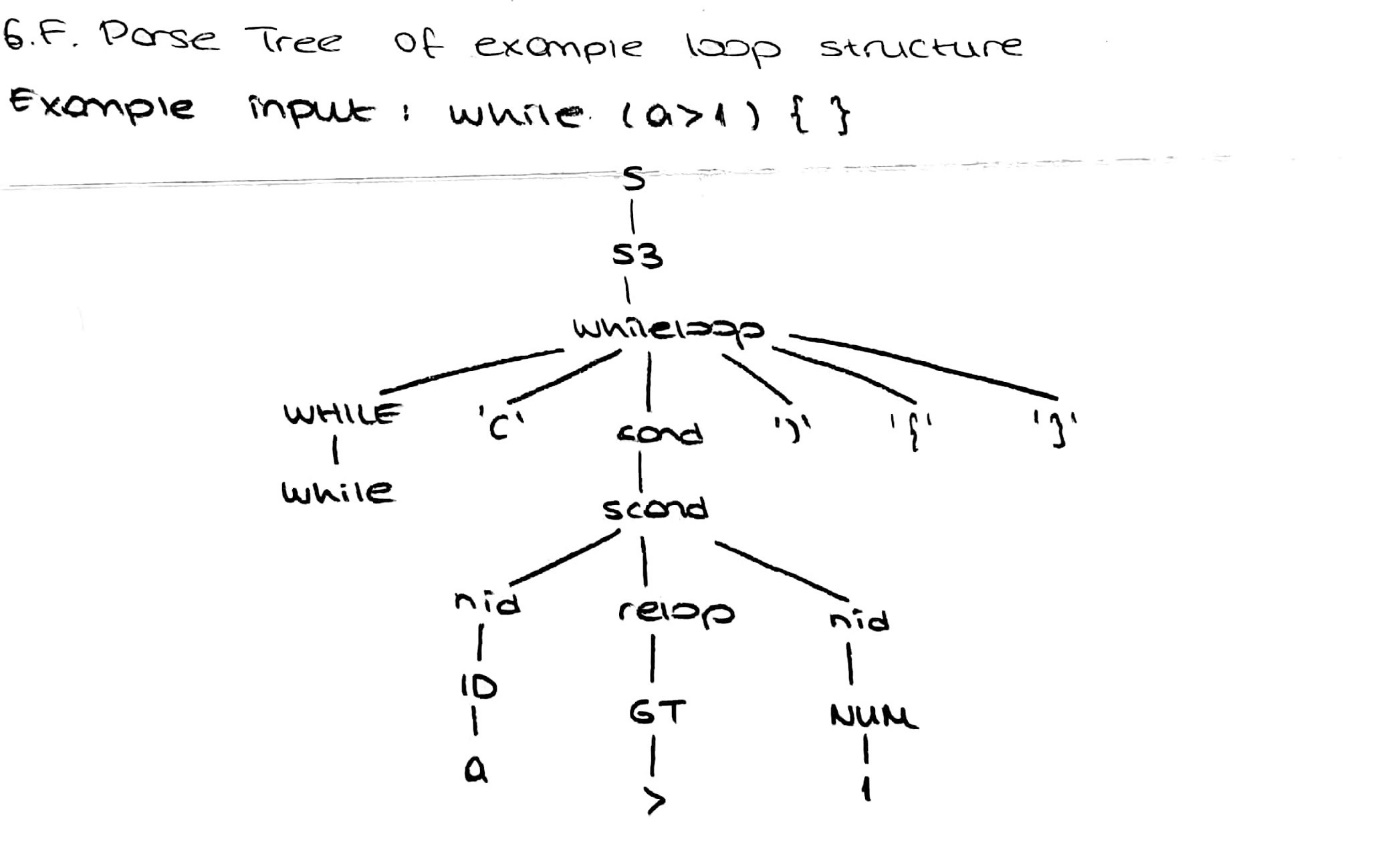
**6.D.** ParseTrees of example comparison operators

****

**6.E.** ParseTrees of example conditional operators

****

**6.F.** ParseTrees of example loop structure

****

**7.**Bibliography

Website that we used in this project:

* <https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.genprogc/yaac_file_declarations.htm>
* <http://publicvoidlife.com/2014/12/18/yacc-program-check-variable-declaration/>
* <http://2k8618.blogspot.com/2011/06/parser-for-if-then-else-statements-yacc.html>
* <https://www.epaperpress.com/lexandyacc/if.html>
* <https://www.humblec.com/yacc-programs-implement-logical-operators/>
* <https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.genprogc/ie_prog_4lex_yacc.htm>
* <https://stackoverflow.com/questions/47420730/linker-error-multiple-definition-of-yylex>
* <https://sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_lib/aixprggd/genprogc/ie_prog_4lex_yacc.htm>
* <https://www.ibm.com/developerworks/library/l-lexyac/index.html>
* <https://medium.freecodecamp.org/the-programming-language-pipeline-91d3f449c919>

***Respectfully,***

***Ayça Nur Vanlı 17290128***

***Özge Çetin 16290086***