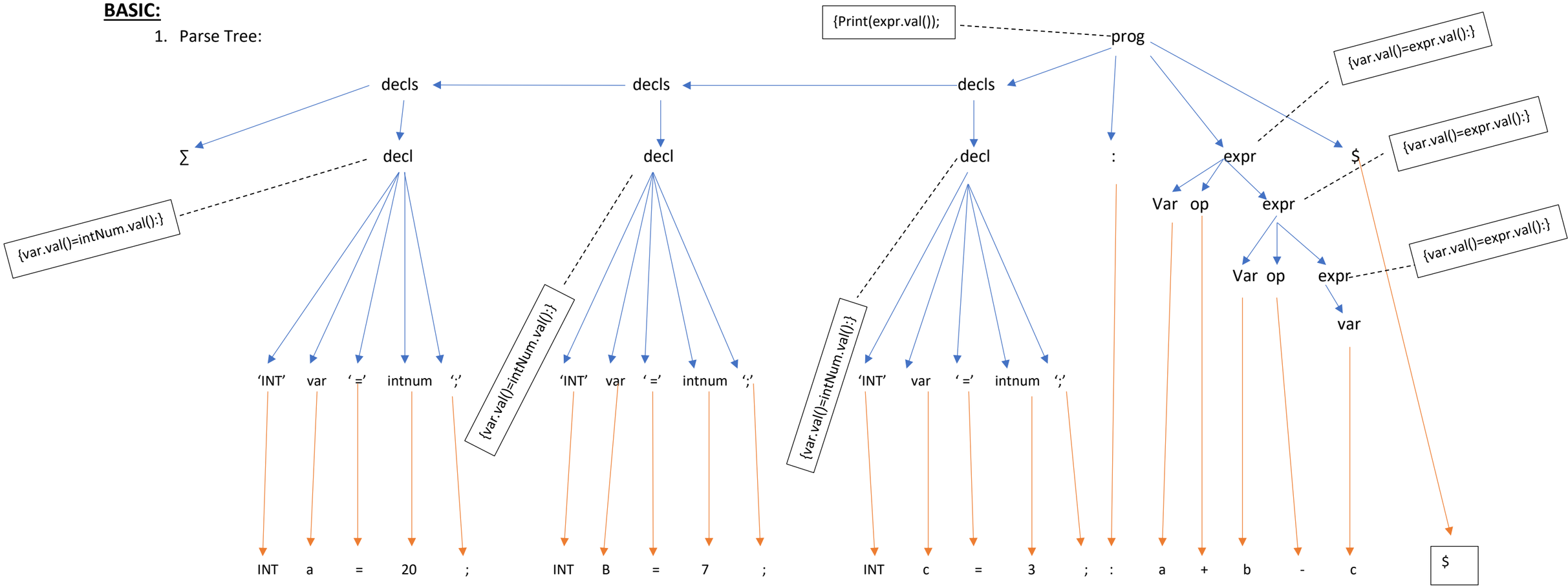
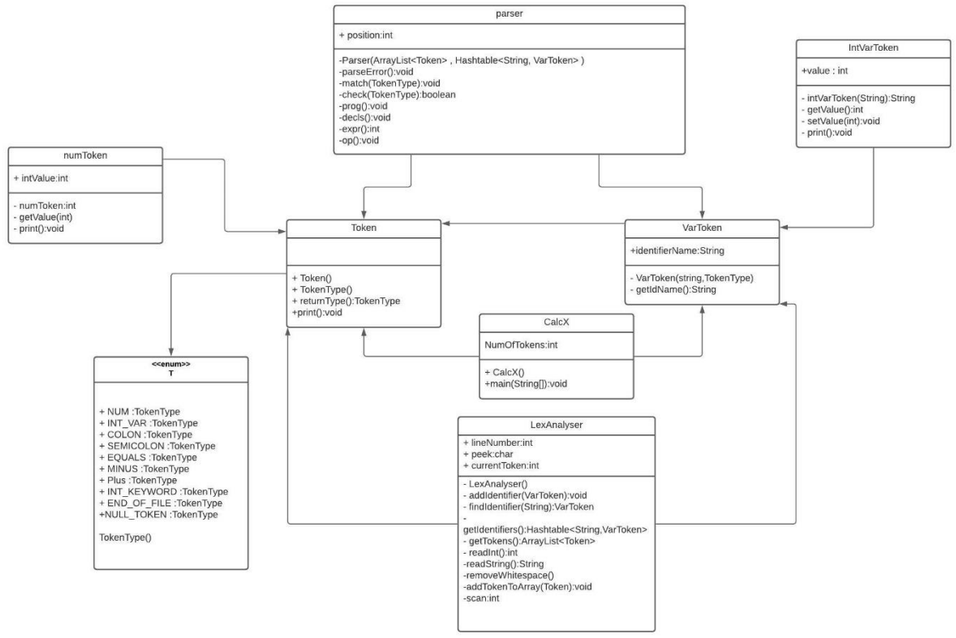
BASIC:

1. Parse Tree:



2. The CFG is right recursive. A production for a non-terminal is recursive if it can derive a sequence containing that non-terminal; it is left-recursive if the non-terminal can appear at the start (LHS) of the derived sequence, and right-recursive if it can appear at the end (RHS). If we look at any of the productions, you would find all of them supporting the previous statements.
   e.g : decls → decl decls
   expr → var op expr

3. Class diagram:



IINTERMEDIATE:

1. I have added a new enumeration for every new operation and added the sign assigning statements for each operation to do its calculation.
   e.g:

```
1. public enum TokenType {
2.     NUM, INT_VAR, COLON, SEMICOLON, EQUALS, PLUS, MINUS, DIVIDE, MULTIPLY, POWER, MODULUS, INT_KEYWORD, END_OF_FILE,
   NULL_TOKEN
3. }
```

```
case POWER:
                op();
                int value2f = expr();
                value = (int) Math.pow(value , value2f);
                break;

```

```
case MODULUS:
                op();
                int value2e = expr();
                value = value % value2e;
                break;

```

CFG:

prog → decls ':' expr '$'

decls → decl decls

| ε

decl → 'INT' var '=' intnum ';'

expr → var op expr

| var

var → [a-z]+

intnum → [0-9][0-9]*

op → '+' | '-' | '^' | '*' | '/' | '%'

SDTS:

prog → decls ':' expr '$' { Print expr.val(); }

decls → decl decls

| ε

decl → 'INT' var '=' intnum ';' { var.val() = intnum.val(); }

expr1 → var op expr2 { expr1.val() = var.val() op expr2.val(); }

| var { expr1.val() = var.val(); }

var → [a-z]

intnum → [0-9][0-9]*

op → '+' | '-' | '^' | '*' | '/' | '%'

2. If condition to check if next token is ':' . If not, the program will match the end of file token by calling the match() function. If the condition is true , the program will get into a while loop to allow the user to enter more than one expression while the condition is true.
   CFG:

```
if (check(TokenType.COLON)) {
                while (check(TokenType.COLON)) {
```

```
3.                      match(TokenType.COLON);
4.                      int newValue = expr();
5.                      i++;
6.                      System.out.println(value + ":" + newValue +":"+ "END");
7.                }
8.         } else {
9.                match(TokenType.END_OF_FILE);
10.               }
11.
```

CFG:

prog → decls ‘:’ expr [‘:’ expr]* ‘$’

decls → decl decls

| ε

decl → ‘INT’ var ‘=’ intnum ‘;’

expr → var op expr

| var

var → [a-z]+

intnum → [0-9][0-9]*

op → ‘+’ | ‘-’ | ‘^’ | ‘*’ | ‘/’ | ‘%’

SDTS:

prog → decls ‘:’ expr [‘:’ expr]* ‘$’ { Print expr.val(); }

decls → decl decls

| ε

decl → ‘INT’ var ‘=’ intnum ‘;’ { var.val() = intnum.val(); }

expr1 → var op expr2 { expr1.val() = var.val() op expr2.val(); }

| var { expr1.val() = var.val(); }

var → [a-z]

intnum → [0-9][0-9]*

op → ‘+’ | ‘-’ | ‘^’ | ‘*’ | ‘/’ | ‘%’

ADVANCED:

-changed the prog() method and added 2 new methods : progB() & progAll().

-progB does exactly as prog but for boolean values not integers :

```
1. public void progB() {
2.
3.         int i = 1;
4.         // First parse declarations
5.         decls();
```

```
6.          match(TokenType.COLON);
7.          String value = String.valueOf(exprB());
8.
9.          if (check(TokenType.COLON)) {
10.                     while (check(TokenType.COLON)) {
11.                             match(TokenType.COLON);
12.                             String newValue = String.valueOf(exprB());
13.                             i++;
14.                             System.out.println(value + ":" + newValue + ":" + "END");
15.                     }
16.             } else {
17.                     System.out.println("Value of expression"+value);
18.                     match(TokenType.END_OF_FILE);
19.             }
20.
21.         }
22.
```

progAll() checks if the program for Boolean or integer declarations & expressions :

```
1.  public void progAll() {
2.              if (check(TokenType.BOOL_KEYWORD)) {
3.                      progB();
4.              }else if(check(TokenType.INT_KEYWORD)) {
5.                      prog();
6.              }
7.
```

exprB applies boolean operations such as AND , OR for declared boolean variables

```
1.  public Boolean exprB() {
2.              Boolean val = true;
3.              if (tokens.get(position).returnType() == TokenType.BOOL) {
4.                      val = Boolean.parseBoolean(((BoolVarToken) tokens.get(position)).getValue());
5.              }
6.              if (tokens.get(position).returnType() == TokenType.COLON) {
7.                      match(TokenType.COLON);
8.                      exprB();
9.              }
10.             match(TokenType.BOOL);
11.             switch (tokens.get(position).returnType()) {
12.             case AND:
13.                     op();
14.                     Boolean value2 = exprB();
15.                     val = val && value2;
16.                     break; // Semantic action
17.             case NOT:
18.                     op();
19.                     Boolean value2b =  exprB();
20.                     val = !val || !value2b;
21.                     break;
22.             case OR:
23.                     op();
24.                     Boolean value2c =  exprB();
25.                     val = val || value2c;
```

```
                    ;

                    break;// Semantic action

            }

            return val;
 }
```

Added 2 classes :

1) BoolVarToken: to read the variable token as a string and store it as a Boolean variable and other methods.
2) BoolToken: checks if declared value of the declared variable is Boolean and other methods.

```
public class BoolVarToken extends VarToken{
   public String value;

   public BoolVarToken(String identName) {
            super(identName, TokenType.BOOL);
            value = null;
   }

   public String getValue() {
            return value;
   }

   public void setValue(String newValue) {
            value = (newValue);
   }

   public void print() {
            System.out.println("Boolean Variable Token: " + identifierName);
   }
public class BoolToken extends Token {
   public Boolean BoolValue ;

   public BoolToken(Boolean value,TokenType T) {
            super(TokenType.BOOLVAL);
                     BoolValue = value;


   }

   public Boolean getValue() {
            return BoolValue;
   }

   public void print() {
            System.out.println("Bool Token: " + BoolValue);
   }
}
```

ADDED a new tokenType enumeration to everything new in the code.

```
public enum TokenType {
     NUM, INT_VAR, BOOL_KEYWORD, BOOLVAL, BOOL, COLON, SEMICOLON, AND, NOT, OR,EQUALS, PLUS, MINUS, DIVIDE, MULTIPLY, POWER, MODULUS, INT_KEYWORD, END_OF_FILE, NULL_TOKEN
}

```

1. CFG:

   progAll→prog | progB

prog → decls ‘:’ expr [‘:’ expr]* ‘$’
progB → decls ‘:’ exprB [‘:’ exprB]* ‘$’
decls → decl decls | declB decls
            | ε
decl → ‘INT’ var ‘=’ intnum ‘;’
declB → ‘BOOL’ var ‘=’ boolnum ‘;’
expr 1→ var op expr 2
          | var
exprB1→ var o exprB2 | var
 var → [a-z]+
 intnum → [0-9][0-9]*
 op → ‘+’ | ‘-’ | ‘^’ | ‘*’ | ‘/’ | ‘%’
 o → ‘&’ | ‘|’ | ‘!’

 SDTS:
 progAll→prog | progB
 progB → decls ‘:’ exprB [‘:’ exprB]* ‘$’ { Print exprB.val(); }
 prog → decls ‘:’ expr [‘:’ expr]* ‘$’ { Print expr.val(); }
 decls → decl decls | declB decls
             | ε
 declB → ‘BOOL’ var ‘=’ boolnum ‘;’
 decl → ‘INT’ var ‘=’ intnum ‘;’ { var.val() = intnum.val(); }
 expr1 → var op expr2 { expr1.val() = var.val() op expr2.val(); }
             | var { expr1.val() = var.val(); }
 exprB1→ var o exprB2 { exprB1.val() = var.val() o exprB2.val(); }
             | var { exprB1.val() = var.val(); }
 var → [a-z]
 intnum → [0-9][0-9]*
 op → ‘+’ | ‘-’ | ‘^’ | ‘*’ | ‘/’ | ‘%’
o → ‘&’ | ‘|’ | ‘!’