Assignment 2

For this assignment, we will write a parser for a subset of a simplified[1] version of the Lua grammar. We'll add the rest of the language in assignment 3.

The subset grammar for this assignment is given below almost[2] as it appears in the Lua Manual. Note that they use different notation than we did in the lecture: instead of A* (i.e. 0 or more instances of A) , they use {A}, and instead of A^+ (0 or 1 instance of A), they use [A].

```
exp ::=  nil | false | true | IntLiteral | LiteralString
       | '...' | functiondef |
        prefixexp | tableconstructor | exp binop exp | unop exp

binop ::=  '+' | '-' | '*' | '/' | '//' | '^' | '%' |
         '&' | '~' | '|' | '>>' | '<<' | '..' |
         '<' | '<=' | '>' | '>=' | '==' | '~=' |
         and | or

unop ::= '-' | not | '#' | '~'

prefixexp ::= Name | '(' exp ')'

functiondef ::= function funcbody

funcbody ::= '(' [parlist] ')' block end

parlist ::= namelist [',' '...'] | '...'

namelist ::= Name {',' Name}

tableconstructor ::= '{' [fieldlist] '}'

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= '[' exp ']' '=' exp | Name '=' exp | exp

fieldsep ::= ',' | ';'

block ::= ϵ
```

The above grammar does not encode the precedence and associativity of operators. The Lua reference manual specifies these separately. Before implementing your

---

[1] We already simplified things for the Scanner by leaving out floating point numbers.

[2] To make this a self-contained subset of the complete languages, a few changes were necessary. For example, the non-terminal `block` only derives ϵ. In a later assignment, we will change that to a list of statements, and also define statements. The production for prefixexp has also been simplified.

parser, you will need to revise the grammar to encode this information. You have seen precedence encoded in grammars in the lecture, and it is also discussed in Scott. Here are the precedence and associativity rules copied from the Lua manual.

> Operator precedence in Lua follows the table below, from lower to higher priority:
>
> ```
> or
> and
> <     >     <=    >=    ~=    ==
> |
> ~
> &
> <<    >>
> ..
> +     -
> *     /     //    %
> unary operators (not   #     -     ~)
> ^
> ```
>
> As usual, you can use parentheses to change the precedences of an expression. The concatenation ('. .') and exponentiation ('^') operators are right associative. All other binary operators are left associative.

Finally, rather than simply parsing to find out whether a sentence is legal, the parse routines will also return an abstract syntax tree. An abstract syntax tree is like a parse tree except that it leaves out irrelevant details like punctuation. The AST reflects the structure of the expression and we should be able to correctly evaluate the expression by traversing the tree and passing the values of subexpressions up to the top. To construct the AST, we define an abstract class Exp which is the superclass of all classes representing expressions. Each different kind of expression is represented by a subclass of Exp. For example, ExpBinary represents a binary expression and has 3 fields: two Exp and one for the operator. The AST classes have been provided for you. Here is ExpBinary, for example. The most important part are the three fields, e0,op, and e1 to hold the two subexpressions and the Token with the operator.

```
package parser.AST;

import scanner.Token;

public class ExpBinary extends Exp {

        public final Exp e0;
        public final Token op;
        public final Exp e1;

        public ExpBinary(Token firstToken, Exp e0, Token op, Exp e1) {
                super(firstToken);
                this.e0 = e0;
                this.op = op;
                this.e1 = e1;
        }

        @Override
        public Object visit(ASTVisitor v, Object arg) throws Exception {
```

```
            return v.visitExpBin(this,arg);
        }

        @Override
        public String toString() {
        return "ExpBinary [e0=" + e0 + ", op=" + op + ", e1=" + e1 + ", firstToken=" + firstToken + "]";
        }

}
```

Ignore the visit method for now. It will be needed in later assignments to implement the Visitor Pattern.

The firstToken field is defined in a superclass. It should hold the first Token in the construct and will be used to locate the beginning of the construct in error messages. When a syntax error is detected, your parser should throw a SyntaxException, which takes a Token and an error message. As before, the contents of your error message will not be graded, but you will be much happier later if the information is useful.

Assuming that your grammar (after modifying to handle associativity and precedence) includes something like

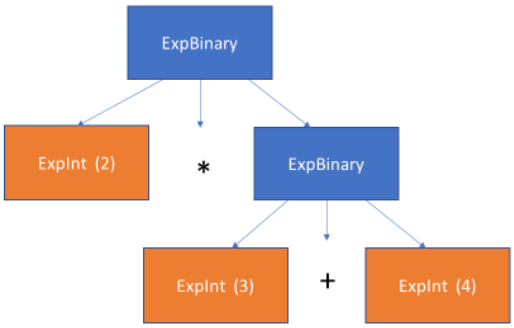
exp :: = andexp { 'or' andexp}

You would have a corresponding method that would return an ExpBinary object. This follows the approach for implementing a recursive descent parser discussed in class. Recall that {..} means 0 or more instances of.

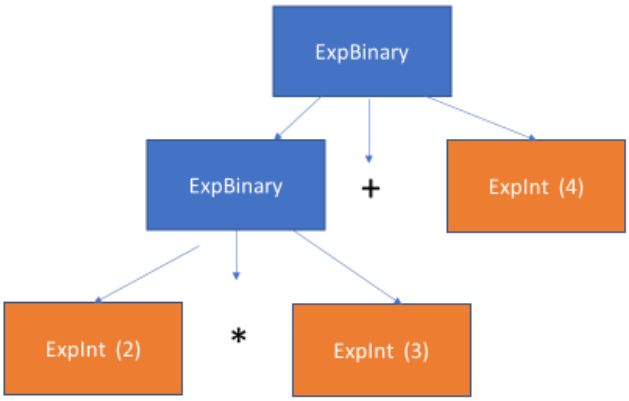
```
Exp exp() throws Exception {
          Token first = t;  Always save the current token when you enter a routine.
          Exp e0 = andExp();
          while (isKind(KW_or)) {
                  Token op = consume();  //consume returns the Token that was consumed
                  Exp e1 = andExp();     //get the second expression
                  e0 = new ExpBinary(first, e0, op, e1);
          }
          return e0;
 }
```

Thus, calling exp() returns an object that is a superclass of Exp. It should be the root of a tree that represents the expression. The shape of the tree should reflect precedence and associativity of operators. Here are a few examples:

2 * (3 + 4)

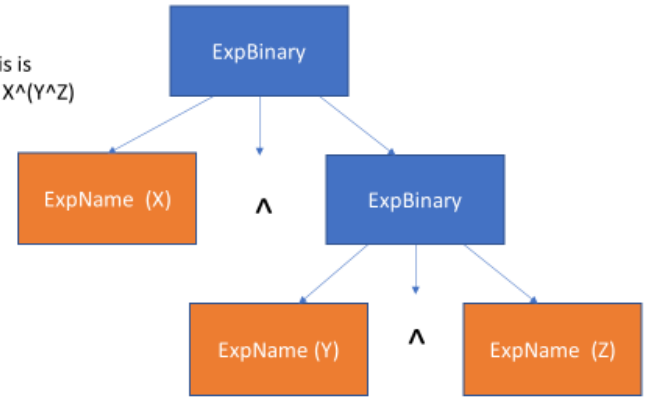

ExpBinary
ExpInt (2)
*
ExpBinary
ExpInt (3)
+
ExpInt (4)

2 * 3 + 4


ExpBinary
ExpBinary
+
ExpInt (4)
ExpInt (2)
*
ExpInt (3)

X^Y^Z

Since ^ is right associative, this is interpreted as X^(Y^Z)



Each production on the R.H.S (with prefixexp replaced by its right hand side) has a corresponding subclass of Exp. These classes have been provided for you and should not need to be modified in this assignment. (You may change the toString method if you want to see more or less information—this would only be used during development.)

```
exp ::=  nil | false | true | IntLiteral | LiteralString
        | '...' | functiondef |
         prefixexp | tableconstructor | exp binop exp | unop exp
prefixexp ::= Name | '(' exp ')'
```

Right hand side of exp ::=	**Type of object to instantiate and return**
nil	ExpNil
false	ExpFalse
true	ExpTrue
IntLiteral	ExpInt
LiteralString	ExpString
'...'	ExpVarArg
functiondef	ExpFunction
Name	ExpName
tableconstructor	ExpTable
exp binop exp	ExpBinary
unop exp	ExpUnary
(exp)	Just return whatever exp does

There are several types of fields with a similar structure: Field is an abstract class, specific types of fields are subclasses.

right hand side of field ::=	**Type of object to instantiate and return**
`'[' exp ']' '=' exp`	FieldExpKey
`Name '=' exp`	FieldNameKey
exp	FieldImplicitKey

The entire grammar is here if you are interested.
https://www.lua.org/manual/5.3/manual.html#9

Turn in

A jar file containing

Scanner.java (from assignment 1, corrected as necessary)

Token.java (from assignment 1)

ExpressionParser.java

All of the AST classes (whether you have modified them or not)

ExpressionParserTest.java containing your own test cases.

A zip file containing a git repository with history from the beginning of assignment 1.

- *Make sure that you are creating a git log file that accurately reflects your development process by committing frequently with descriptive log messages.*
- *Do NOT use the github download zip function to create your zip file. It does not include the history, defeating the purpose of this requirements.*

Hints:

The Token class now has an equals method that does not include the line or position. This makes it easier to devise test cases.

The Expressions class contains several factory methods for creating instances of AST nodes. These are useful for testing as they can be directly compared with AST nodes constructed by the parser.

During development, I find it is more useful to let not-yet-implemented methods throw an UnsupportedOperationException rather than return null. This is just a suggestion. Your final submission will not have any unimplemented methods.