Assignment 3

For this assignment, we will write finish the parser for the subset of Lua that we will consider this semester. The grammar shown below includes the grammar from assignment 2. New productions are in blue. With the exception of prefixexp, and block, the productions that were given in assignment 2 remain the same. The prefixexp production has an additional alternative.

The subset grammar for this assignment is given below almost as it appears in the Lua Manual. Remember 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].

```
chunk ::= block

block ::= {stat} [retstat]

stat ::=  ';' |
        varlist '=' explist |
        ~~functioncall |~~
        label |
        break |
        goto Name |
        do block end |
        while exp do block end |
        repeat block until exp |
        if exp then block{elseif exp then block}[else block]end|
        for Name '=' exp ',' exp [',' exp] do block end |
        for namelist in explist do block end |
        function funcname funcbody |
        local function Name funcbody |
        local namelist ['=' explist]

retstat ::= return [explist] [';']

label ::= '::' Name '::'

funcname ::= Name {'.' Name} [':' Name]

varlist ::= var {',' var}

var ::=  Name | prefixexp '[' exp ']' | prefixexp '.' Name

namelist ::= Name {',' Name}
```

```
explist ::= exp {‘,’ exp}

exp ::=  nil | false | true | IntLiteral | LiteralString
       | ‘...’ | functiondef |
        prefixexp | tableconstructor | exp binop exp | unop exp

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

unop ::= ‘-’ | not | ‘#’ | ‘~’

prefixexp ::= var | functioncall | ‘(’ exp ‘)’

functioncall ::=  prefixexp args | prefixexp ‘:’ Name args

args ::=  `(´ [explist] `)´ | tableconstructor | String

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 ::= ‘,’ | ‘;’
```

This version of the grammar needs some modification in order to be LL. Hint: start with prefixexp and replace var and functioncall with their right hand side. Eliminate left recursion (how to do this was described in the lecture). Left factorize as necessary. There will be one situation where the grammar is still not LL(1). You can handle that as a special case. (This was discussed in class.)

An expr that ends with (..) should create an ExpFunctionCall object. An expr that ends with [,,,] should create an ExpTableLookup object. Note that function application and table lookup associate to the left. So something like f(x)[a] would create an ExpFunctionCall object e, where f is the expression designating the function and an argument list containing the Expression representing x. Then there would be an ExpTableLookup object with e as the expression and an expression representing a as the key.

Syntactic Sugar:

a.name is syntactic sugar for a["name"], and should be treated as a["name"] for the purpose of creating the AST node.

A call `v:name(`*`args`*`)` is syntactic sugar for `v.name(v,`*`args`*`)`, except that `v` is evaluated only once. This can be handled by treating v.name as a table lookup as above, and adding a reference to the expression representing v to the front of the argument list.

Turn in

A jar file containing

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

Token.java (from assignment 1)

Parser.java

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

ParserTest.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. Feel free to add additional methods if useful, but don’t remove or change the ones that are given.

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 should not have any unimplemented methods.

Create a new git branch before adding the new AST nodes.