In this assignment, we will implement a recursive interpreter to handle a subset of our language. This will require implementing two ASTVisitor classes. One, called SemanticAnalysis will traverse the AST, collect static information, and annotate some of the AST nodes with static information that will be useful later. (In a more efficient interpreter, we could generate most of this data during parsing, but it is easier to manage to do it separately.) The other one traverses the AST, evaluating expressions and executing statements.

Our language, following Lua, is dynamically typed. This means that variables do not have types, but values do. This means that at runtime, variables must be represented in a way that allows the type to be determined. In the actual Lua implementation (which is written in C), the following struct is used

```
typedef struct {                typedef union {
  int t;                          GCObject *gc;
  Value v;                        void *p;
} TObject;                        lua_Number n;
                                  int b;
                                } Value;
```

Without going in to the details, you can see that the TObject has a field t representing the type, and a field with the Value. The value is the union which can hold any of the types in Lua, some of which are pointers.

In our implementation, we will use the object-oriented features of Java to handle this situation. LuaValue will be the base class, and the various types will be subclasses. These include LuaInt, LuaString, LuaTable, and LuaNil. There should only be one instance of LuaNil, which is LuaNil.nil. Two more types of LuaValues are LuaFunction and JavaFunction. For this assignment, we will not handle LuaFunctions, so they are not included. JavaFunction is an abstract class; a function written in Java that can be called by our language should provide a concrete subclass that implements the call method. (Three such JavaFunctions have been provided; you probably won't need any more.) The LuaTable implementation is similar to the one described in
http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html

Your implementation must support all of the above mentioned LuaValues except LuaFunction.

Your interpreter only needs to handle a subset of the language that we have implemented so far. You do NOT need to handle local variable, and you do not need to handle function calls of functions expressed in our language. You DO need to be able to call JavaFunctions (which must satisfy the given interface). Since we will not have local variables, all variables are global. We will also not handle variable length function arguments.

Handling most language features is fairly straightforward in your interpreter. The visit method for an expression evaluates the expression and returns its value. The statements implement the appropriate semantics. For example, the visitStatAssign method visits the right hand side expressions to get their values, then assigns the values to a global variable or table fields as appropriate. Situations where the number of right hand side expressions differs from the number of targets on the left hand side should be handled as done in Lua (see https://www.lua.org/manual/5.3/manual.html#3.3.3) Global variables are stored in a LuaTable that is created on initialization of the Interpreter.

The statements that you need to handle are Blocks, Chunks, Assignment, If, While, Repeat, Break, and Goto. You do NOT need to implement For Statements, Local Declarations, or Function Calls as Statements. Of these, the only ones that are not straightforward are break, goto and label, and return. Break and goto involve jumping out of the current block or loop. The exception handling facilities of Java will help here.

The goto statement needs additional preparation. The Lua manual states: "A label is visible in the entire block where it is defined, except inside nested blocks where a label with the same name is defined and inside nested functions. A goto may jump to any visible label as long as it does not enter into the scope of a local variable." The visibility of labels is similar to the visibility of variables in block structured languages that we discussed in class. To prepare the AST for goto statements, create an ASTVisitor called StaticAnalysis that traverses the AST and annotates every goto statement with its target label. Note that there may be more than one label with the same text, your StaticAnalysis needs to choose the correct one. It is also possible that a goto statement is trying to jump to a label that isn't visible or doesn't exist. In this case, an exception with appropriate error message should be thrown.

The jar file contains a class InterpreterStarter and a class InterpreterTest that illustrate how to set up the interpreter. A fairly extensive set of test cases has been provided, but they are not complete. In particular, you need to implement binary and unary operators that are not tested in these tests.

Turn in

A jar file containing

Source code of all the classes necessary to implement the interpreter as specified.

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 requirement.*

Note: some of the AST nodes have been modified, either by adding fields need in this assignment, or tweaking the toString method. You will probably want to put the new code in a new git branch and then merge with your previous code.