1. Design: Our design has three folders' worth of classes:
    - Intermediate Representation nodes (IR*), which are fairly directly converted from corresponding statements in code, and store tables for any fields or methods defined within them; and
    - Symbol Tables, which are hashmaps from the string name of a variable or method to its IR node. These allow easy lookup of methods or fields that are accessed, and automatically search in their parent tables if they don't have a corresponding method/field in them. These are ProgramTable, ClassTable, MethodTable, and VariableTable. VariableTable generically handles fields, parameters, or locals. ClassTable is never used except as a parent class for ProgramTable since decaf does not have classes, but we wanted to have the symbol table capacity to include classes; and
    - a "trees" folder where we have code that converts Antlr's output into something that's actually a tree (ConcreteTree), and then transforms that into an Abstract Syntax Tree out of the IR nodes and Symbol Tables, and then has a SemanticChecker and EnvStack to do checks on it.

   The SemanticChecker is a separate class that can be instantiated to build a stack of environments, and run on a program AST. It has roughly one check method for each IR class we built. We considered using the visitor pattern but decided that was too complicated to learn and implement quickly. The environment stack is an object containing a number of different types of things: a stack of variable scopes, a stack of method scopes, a stack of for/while loops that can be broken out of, and a stack of what return types are expected.
2. Extras: As of now, there have not been any substantial extras used in our project. We did write a class symbol table even though classes are not part of decaf, because we wanted to understand how to build the tables to make them work, but it's not integrated into the rest of our system.
3. Difficulties: We had difficulty with the type system. We want every intermediate expression to be able to return something from getType(), but this often requires looking up the type of a variable or looking up the return type of a method, and we don't necessarily have the symbol tables connected to those nodes. So far, we got around this by having our semantic checker set the types of things as it encounters them, which is maybe a reasonable approach because it's checking that the semantics are such that every expression can be assigned a consistent type. This feels inelegant since it requires the semantic checker modifying the tree, though. In general we have significant refactoring remaining to do.
4. Contribution: Arkadiy mostly designed the IR nodes, Jackie built the symbol tables and integrated them into the AST, and Maya wrote the code to build the ConcreteTrees and ASTs. Jackie then wrote the SemanticChecker file, and Maya completed some of its methods. Jackie and Arkadiy ran these semantic checks on test inputs and iteratively changed things to debug, and wrote this documentation.