6.035 Project 2: Semantic Checker Write-up
Group: le03

1. A brief description of how your group divided the work. This will not affect your grade; it will be used to alert the TAs to any possible problems. (Projects 2 through 5 only.)

Maksim:
- Statement classes for the AST
- Descriptors
- ArraySizeCheckVisitor
- BreakContinueStmtCheckVisitor

Dhruv:
- Generation of AST from ANTLR Parser
- SymbolTableGenerationVisitor
- MethodCheckVisitor

Kainar:
- IntOverflowCheckVisitor
- Internal test cases

Usman:
- Everything else

2. A list of any clarifications, assumptions, or additions to the problem assigned. The project specifications are fairly broad and leave many of the decisions to you. This is an aspect of real software engineering. If you think major clarifications are necessary, consult your TA.

Assumptions: No same unary operators are next to each other without parentheses. For example --1 is not a valid statement, but -(-1) is valid.

Clarifications: There is one class called Program. Method overloading is not allowed in Decaf.

3. An overview of your design, an analysis of design alternatives you considered, and key design decisions. Be sure to document and justify all design decisions you make. Any decisions accompanied by a convincing argument will be accepted. If you realize there are flaws or deficiencies in your design late in the implementation process, discuss those flaws and how you would have done things differently. Also include any changes you made to previous parts and why they were necessary. A brief description of interesting implementation issues. This should include any non-trivial algorithms, techniques, and data structures. It should also include any insights you discovered during this phase of the project.

Abstract Syntax Tree generation from ANTLR Parser
The first decision we made was to use heterogeneous syntax trees instead of homogeneous syntax trees. Heterogeneous trees makes the Parser.g file more messy, but ultimately provides more flexibility and control. We had to make modifications to

Parser.g by adding actions to each production, which created the nodes of the AST. To correctly work with the left recursion in the Parser, we had to create a TempExpression node which behaved like a Temporary Expression. This node was never in the final AST, but acted like a placeholder for content it was consolidated into an Expression node. An issue which we realized later in the process was that we need a mechanism to keep track of line and column numbers. We added the line and column getters and setters to the generic AST class. In Parser.g, we added a private class called StringToken, which acted as a temporary container for token values, line number, and column number.

Descriptors
Descriptors are generated by going through the AST and retrieving the information we need for low-level IR. We made one significant design decision: descriptors and AST nodes are agnostic to each other and they don't share any interfaces, so only primitive data is transferred from AST node when creating a Descriptor.

Symbol Tables
One thing we do differently from what was told in lecture is the way we organize the scope hierarchy. The scope table is a global hashmap used to keep track of the scopes. The table maps unique block ID's to the local GenericSymbolTable associated with those blocks (scopes). Each GenericSymbolTable has a pointer to the GenericSymbolTable of the parent scope. At the method scope level the parent pointer points to the parameter symbol table of the method and the parent pointer of the parameter symbol table points to the field symbol table of the class. One related design decision is that we use a single class, GenericSymbolTable, for storing local, parameter, and class field symbols instead of creating different types of symbol tables for each kind of variable.

Other than the GenericSymbolTable, we have a MethodSymbolTable which matches method names to MethodDescriptor objects.

Visitor Pattern
The general idea is to recursively visit the tree of AST nodes (starting with the node representing the single Program class) and add semantic check functionality to each node in the AST. Different visitors check different functionality. We go over each visitor below:
- General design decision regarding whether or not to abstract out the recursive logic for each node vs. repeat the recursive logic in every visitor

MethodCheckVisitor
Checks for method signature matching in method calls. Ensures that methods return the right type. Ensures that if a method returns a non-void type, every logical termination point in the method ends with a return statement.
In the initial attempt we didn't make use of the return value of the nodes in the AST when checking whether every logical block returned the desired RETURN type; as a result, we did recursion within a visit to a single node which was not taking advantage of the tree structure. Now, we return a boolean instead of int and use that value to make the RETURN statement check cleaner.

IntOverlowCheckVisitor

Checks if integers that were assigned are within valid boundaries. Negative numbers can go up to -2147483648, while positive up to 2147483647. If we find a unary expression with an int literal child, we replace the sub-tree rooted at the unary expression with an int literal with a negative value.

BreakContinueStmtCheckVisitor
Checks if all the break and continue statements are inside the for loops. The visitor maintains a boolean state inFor, which is initially set to false. Whenever we enter a for statement, the state is set to true and is set back to false when we are out of the for statement. When the visitor visits a break or continue statement, it just checks if the state is set to true and if not, returns an error.

ArraySizeCheckVisitor
Checks if the sizes of the arrays are greater than 0. When the visitor visits an array field, it simply checks if the value of the array length of that field is larger than 0 and if not return an error.

SymbolTableGenerationVisitor
This visitor goes through the AST and generates SymbolTables. While generating the SymbolTables, it also does reference checking to make sure all identifiers have been instantiated at some scope in or above the current scope. The SymbolTables are prepared for future visitors to leverage for their own semantic checks. We use a global variable to keep track of the current scope so we are updating the right SymbolTable; we update the current scope as we recurse up or down the tree.

TypeEvaluationVisitor
The TypeEvaluationVisitor evaluates type of every Expression object. While evaluating types of expressions it also type checks to ensure that the operands of binary and unary expressions have the right types. It also checks assignment statements for correct semantics. All semantic checks from numbers 10-17 are done by this visitor. While writing this visitor, one flaw which we realized that since we never specifically distinguished between array and non-array Type at the AST level. This made things complicated at the Visitor level. We fixed this by refactoring the Type enum to distinguish between INT and INTARRAY, BOOLEAN and BOOLARRAY

4. A list of known problems with your project, and as much as you know about the cause. If your project fails a provided test case, but you are unable to fix the problem, describe your understanding of the problem. If you discover problems in your project in your own testing that you are unable to fix, but are not exposed by the provided test cases, describe the problem as specifically as possible and as much as you can about its cause. If this causes your project to fail hidden test cases, you may still be able to receive some credit for considering the problem. If this problem is not revealed by the hidden test cases, then you will not be penalized for it. It is to your advantage to describe any known problems with your project; of course, it is even better to fix them

All provided test cases pass. We noticed some issues with how line and column numbers were being printed. Specifically, these numbers were corresponding to the actual location of the error in some cases. However, we have fixed most of these ambiguities to the best of our abilities.