CSC488 Assignment 3

Vishrant Vasavada, Tian Ze(Peter) Chen, Anthony Vandikas, Felix Lu, Badrul Chowdhury
25th February, 2015

Overview

The starter code guided us in generating the AST structure during parsing. For semantic analysis, we created four classes in src/compiler488/semantics/ to represent scopes (MajorScope.java and MinorScope.java) and the results of semantic analysis (ExpnSemantics.java and StmtSemantics.java). We implemented semantic analysis by augmenting the AST.

Symbol Table

We represent our symbol table (src/compiler488/symbol/SymbolTable.java) as a stack of scopes, where each scope is represented by a map from an identifier to a symbol. Our symbol table has methods to push new scopes, pop existing scopes, lookup entries, and add entries to the current (topmost) scope.

A symbol (src/compiler488/symbol/Symbol.java) contains only type information. As identifiers can refer to things that are not scalar values (i.e. integers or booleans), our symbol class represents an extended set of types. This set includes array, function, and procedure types, as well as the basic scalar types.

AST Design

Our AST was mostly based on the starter code provided. All nonterminals generate an AST node. For example, the assignment statement is generated as such:

```
variable:v LESS:l EQUAL expression:e
     {: RESULT = new AssignStmt(lleft, lright, v, e); :}
```

Notice how the AssignStmt constructor takes two extra arguments, lleft and lright. We have augmented the AST interface with line and column numbers, and have updated all the relevant constructors accordingly:

```
public AssignStmt (int line, int column, Expn  lval, Expn rval)
```

We have augmented our AST with a few more methods, which are described below.

Semantic Analysis Design

Expressions and statements have a checkSemantics method that performs semantic analysis and records errors as needed. The checkSemantics method takes in all information that may be required for semantic analysis, and returns all information that the parent node may require for semantic analysis. In particular:

- Expression checking requires information about the symbol table and the major scope. Expression checking returns the expression type (boolean or integer) as well as a boolean that indicates whether the expression would cause a function or procedure to return.
- Statement checking requires information about the symbol table, major scope, and minor scope. Statement checking returns a boolean indicating whether the statement would cause a function or procedure to return.

Error Reporting

Since our AST interface has been augmented with line and column information, we use this information to enhance our error messages. We have augmented the BaseAST class with a createError method that creates creates an informative and nicely formatted error message.

Type Tracking

As mentioned above, types are tracked through the return value of checkSemantics for expressions. If semantic analysis fails for a particular AST node in such a way that the expression type cannot be determined, then the return type is null. It is up to the caller of checkSemantics() to detect null types. Statements do not have a type, so the StmtSemantics class does not have a corresponding field.

Difficulties Faced, Conclusion and Special Credits

The most difficult part of the assignment was writing the test case generator. Generating semantically correct programs is a more difficult problem than simply generating syntactically correct programs, as we did in the last assignment. However, writing a correct generator greatly improved our understanding of the semantic rules and illuminated the required steps to implement semantic analysis. Our generated tests caught a number of bugs during the development of the semantic analyzer, and we consider the generator to be worth the time it took to create it.