CSC488
ASSIGNMENT 3
SEMANTIC ANALYSIS & SYMBOL TABLE

DANIEL BLOEMENDAL

Contents

1. Overview

While planning the design for semantic analysis, there were a number of alternative approaches to consider. One suggested approach was to implement all semantic analysis logic in the AST nodes themselves. Another approach was to implement some type of visitor pattern, as suggested in, `CSC488H1S_ast.pdf`. Ultimately, I decided to consolidate all semantic analysis in the single class `Semantics` in `Semantics.java`. The rationale behind this was to avoid a haphazard implementation plan and opt for a thorough systematic approach. Each semantic action and language construct was handled one after another in a single file, with code which is both easy to read and understand. Another critical advantage of the approach is the ease with which it was possible to verify that the semantic analyzer covers all of the language grammar and each semantic action by a brief visual inspection of the code.

2. Processors

At its core, the semantic analyzer is based on a depth-first traversal of the AST. During traversal, as each node is visited a set of *processors* have the opportunity to inspect the visited node and call the various semantic actions. Each processor is bound to some target AST class. As a node is visited, any processor targeting the class of the visited node is fired. Furthermore, there are two types of processors, *preprocessors* and *postprocessors*. Preprocessors are fired when an AST node is seen for the first time. Postprocessors are fired when an AST node is seen for the second and last time, after all of the node's children have been processed.

The processors are implemented using Java annotations and reflection. A processor is defined by annotating a function with the `@PreProcessor` or `@PostProcessor` annotation with the target AST class as an annotation argument. Below is an example of a processor.

```
/*
 * expression S31 '+' expression S31 S21
 * expression S31 '-' expression S31 S21
 * expression S31 '*' expression S31 S21
 * expression S31 '/' expression S31 S21
 */
@PostProcessor(target = "ArithExpn")
void postArithExpn(BinaryExpn binaryExpn) {
    setTop(binaryExpn.getLeft());
    semanticAction(31); // S31: Check that type of expression or variable is integer.
    setTop(binaryExpn.getRight());
    semanticAction(31); // S31: Check that type of expression or variable is integer.
    semanticAction(21); // S21: Set result type to integer.
}
```

At startup, these processors are then stored in a map from AST class name to function. This is done via reflection in the `populateMappings` routine in `Semantics`. Using reflection, all the methods in the `Semantics` class are enumerated and checked for annotations. If our annotations are found tied to a method the method is then stored in the appropriate map. The maps are then used during AST traversal to call appropriate processors as each node is visited.

3. Actions

As each processor is executed during AST traversal, those processors in turn call on semantic actions to perform the necessary semantic checks outlined in the given semantics documentation, `CSC488H1S_semantics.pdf`. Actions are also implemented using annotations and reflection in a manner almost identical to processors. This time, an action is defined by declaring an action function annotated with `@Action` with the semantic action number as the argument passed to the annotation. An example of an action is provided below.

```
@Action(number = 46) // Check that lower bound is <= upper bound.
Boolean actionCheckArrayBounds(ArrayDeclPart arrayDecl) {
    ArrayBound b1 = arrayDecl.getBound1(),
               b2 = arrayDecl.getBound2();
    if (arrayDecl.getDimensions() >= 1 && b1.getLowerboundValue() > b1.getUpperboundValue()) {
        setErrorLocation(b1); return false;
    }
    if (arrayDecl.getDimensions() >= 2 && b2.getLowerboundValue() > b2.getUpperboundValue()) {
        setErrorLocation(b2); return false;
    }
    return true;
}
```

As previously mentioned, the actions are called on by processors as they operate on the various nodes of the AST. They are called through the `semanticAction` function in `Semantics`. The `semanticAction` function looks up the function corresponding to the semantic action number and invokes it, passing to it the AST node currently being visited by the processor. It should be mentioned that processors have the opportunity to change the target AST node that is being inspected before every semanticAction call via the `setTop` function.

The actions have the opportunity to perform either some internal bookkeeping or a semantic check. If an action performs a semantic check it returns whether the check succeeded or failed via the return value of the action function. If an action returns a failure the `semanticAction` function will report the failure. It calls on my team mate Peter's pretty printing facilities to accurately display the location in source code of the failure, by passing it the AST node where the failure occurred.

4. Symbol Table

The design of the symbol table is fairly straight forward. The symbol table stores a stack of scopes. Each scope has a map from identifier to symbol. A symbol object is either an instance of `VariableSymbol` or `FunctionSymbol`. Each symbol stores information about a declared variable or function. The `VariableSymbol` class includes support for 1D and 2D array variables.

```
class Scope {
    SymbolTable.ScopeType type;
    Map<String, Symbol>   map;
    RoutineDecl           routine;
    ...
}
```

It should also be mentioned that each scope also has a type, which currently isn't used, specifying which type of scope it is, be it a program, function, procedure or statement scope. Also, it is possible to associate a scope with a routine. This is useful during semantic analysis, as it can be used to tie statements to their enclosing routines.

The `SymbolTable` class has exposes functions to enter and exit scopes, `scopeEnter` and `scopeExit` respectively. The semantic analysis component can then enter and leave scopes and call `scopeSet` to associate an identifier with a `Symbol` instance. As for retrieval, it has an interface, `find`, to search for a visible symbol by walking up the stack of scopes. It should be noted that `find` also supports searching the current scope only.