CSC488 Assignment 3

Design Document

Jasmeet Brar, Kareem Golaub, Kyokeun Park, Jasmeet Sidhu

AST Design

With the AST containing "just enough" of an implementation to represent all possible programs, it's up to us to add/reorganize the AST hierarchy to make it more amenable for semantic analysis.

The AST is built for semantic analysis by having a stack run in parallel to the parse stack, accessed via `RESULT` and colon tags in JCUP (see `csc488.cup`, e.g. `RESULT = new ArrayDeclPart(name, b);` for a single bound array, or `RESULT = new ArrayDeclPart(name, b1, b2);` for 2D array), which allows us to construct corresponding objects seen in `compiler488/ast/decl, expn, stmt` etc.

We opted to then have this AST be "visited" by our external semantic analyzer. Source coordinates are not implemented (this is interpreted as acceptable by #78 on Piazza). Notable additions/changes to `compiler488/ast` include (U for update, A for added):

- U **BaseAST.java**
  Addition of `accept` method for our semantic analyzer visitor. Corresponding subclasses e.g. U `compiler488/ast/decl/*`, U `compiler488/ast/expn/*` etc. were updated to override this method, determining how to visit its descendants.
- A **ReadableExpn.java**
  Parent class for expressions meant to be represented as AST nodes that can be an argument in a GET statement (extends `Readable`).
- U **Program.java**
  Updated with a new constructor that takes in a `Scope` and replicates its contents.

Symbol Table Design

To determine the validity of the declaration and use of variables, functions and procedures, we needed to keep track of the relevant type and scope of all these symbols as we analyzed the tree. Notable additions/changes include:

- A **ScopeNode.java**
  Represents an instance of a scope. Contains a reference to its corresponding `SymbolTable`. Contains the actual map from identifiers to different types of symbols (see `Record`). Has access to its parent scope, facilitating traversal for outer symbols, and has optional fields for the scope's type and "parameters" (if applicable). We also have a field named archive which will be explained in `SymbolTable`'s `exitScope()`.
- A **Record.java**

Represents a symbol and symbol metadata, such as `RecordType` (i.e. type of symbol: scalar, procedure, function, parameter, array), "return type", and list of "parameters" (if applicable).

- A **Context.java**
  `Context` has a dual-purpose: 1. to track the current major scope (i.e. don't change major scope unnecessarily when encountering minor scopes, except loops) being analyzed, and 2. to facilitate loop and return counting. It contains a `ContextType` denoting, the number of main, procedure, function or loop, the number of returns in the scope, and the number of loops created so far, which we find useful for semantic analysis.

- U **SymbolTable.java**
  Our symbol table implementation consists of a single `ScopeNode` reference and functions operating on that `ScopeNode`. Notable functions are:

  - **get(String key)** Given a key value, go through the `ScopeNode` and its parents to attempt to retrieve the value linked to the key. If there are no reference to the key, then the program will print an error.
  - **put(String key, Record value)** Given a key and value, stores the combination in the current `ScopeNode`'s hash table. If there is a same key in the hash table already, then assert.
  - **enterScope()** Called everytime semantic analyzer notices a new scope. We will create a new `ScopeNode` object and link the previous `ScopeNode` as its parent.
  - **exitScope()** Called everytime semantic analyzer notices that it is leaving the scope. We will set current `ScopeNode`'s parent as the new current `ScopeNode` (root). Furthermore, we will store the previous `ScopeNode` in archive. This may not be necessary at the moment, but may come in handy if we need to traverse the AST again, or need to generate a file out of it.

Semantics Design

We updated `Semantics.java` and created `ASTVisitor.java` to utilize the visitor pattern for semantic analysis.

- A **ASTVisitor.java**
  Represents a visitor to the AST, being able to visit each possible type of AST node.

- U **Semantics.java**
  An AST visitor. Added checks for each semantic action described in `semantics.pdf` here. Then override each visit action on a node type with the correct sequence of calls to the semantic actions. Each check prints an error for the action if it's requirements aren't met. These checks utilize metadata described under `Symbol Table Design`.

  We note that the semantic checks for actions 2, 3, 13, 15, 16, 22, 23, 26, 29, 33, 39, 44, 45, 48 and 49 are missing. This is because these checks were implemented implicitly while constructing the AST or implicit in the implementation of other actions.

Testing

All the test cases used are described in `test/Readme.tests`. We constructed two sets of test cases to test our semantic analyzer:

- Failing test cases to test our analyzer's ability to detect incorrect semantics
- Passing test cases to test our analyzer's ability to pass correct semantics

The passing test cases were divided into two categories:

- AST test cases: cases that would test our analyzer's ability in getting the correct AST's
- Symbol table test cases: cases that would test our analyzer's ability in analyzing the source code, and generating the correct symbol table

To run the tests, after compilation, execute: `./RUNTESTS.sh`

As for how the test cases determine our analyzer's ability:

In each of the failing test cases, we have source code that would contain some case of incorrect semantics. When the analyzer goes through the source code, it visits each part of the whole AST, and attempts to call the appropriate semantic action for the respective part. Each semantic action has some check to see if that part is semantically correct. If it isn't semantically correct, which is the case for all the failing test cases, a flag is set and a message would be sent back by the analyzer, saying: "Exception during Semantic Analysis". So for our failing test cases, the script would check if the analyzer is returning the message back after it analyzed the test cases, which would indicate that the analyzer was able to detect incorrect semantics.

As for the passing test cases, the source code would be semantically correct, so when the analyzer goes through the test cases, it wouldn't return any error message, and this would indicate that the analyzer determined that there are no incorrect semantics with the test cases, which is correct.

Simply testing whether or not the analyzer is able to check that the source code is semantically correct isn't enough; we also need to check if the analyzer has actually determined the AST correctly for each part of the source code, and it isn't just emitting that there are no errors. So for the AST test cases, the test script would have the analyzer go through all the .488 source files in `passing/ast` directory, and output their respective overall AST. Then the script COMPARES this to their respective .out file, which contains the correct AST, which we verified, that should be outputted by the analyzer. If there are no differences, then the test script would deem that the analyzer has passed the test case, which indicates that the analyzer was able to determine the correct AST.

Another important component that we needed to test is the symbol table being generated by the analyzer, since we needed to make sure that the analyzer is correctly determining the scopes in the source code, and the variables that are in the scopes. To do this, the script would have the analyzer go through all the .488 sources files in `passing/symbol_table`, and dump their respective symbol table. Then the script would then compare this to their respective .out file, which contains the correct symbol table, which we verified, which should be outputted by the

analyzer. If there are no differences, then the test script would deem that the analyzer has passed the test case, which indicates that the analyzer was able to generate the correct symbol table.

Contributors

- Jasmeet Brar - created test cases, documentation, semantic analyzer design discussion, code review
- Kareem Golaub - created semantic actions, semantic analysis bug fixes, documentation, code review
- KyoKeun Park - implemented semantic analyzer, implemented symbol table, created semantic actions, semantic analyzer design discussion, code review
- Jasmeet Sidhu - augmented the AST for semantic analysis, created semantic actions, semantic analysis bug fixes, created test cases, created testing script, semantic analyzer design discussion