Semantic Analysis Design

Tatum Alenko

March 31, 2019

Table of Contents

Contents

Section 1. Analysis

Checklist of implemented semantic checks

- ☒ 1: Global scope symbol table
- ☒ 2, 3, 4: Nested scope symbol tables for each 'scope type' (linked through recursive structures)
- ☐ 5: During symbol creation, some semantic errors are declared and reported, such as ...
- ☒ 5.1: Multiply declared identifiers in same scope
- ☐ 5.2: Shadowed inherited members
- ☐ 6: Undefined member functions and free functions
- ☒ 7: Symbol table text representation outputted to a file for demonstration
- ☒ 8: Mutiply class and variable identifiers in the same scope
- ☐ 9: Function overloading
- ☐ 10: Type checking on all expressions
- ☒ 10.1: Type checking on some expressions (such as index lists, assignment statements, variable element lists, data members)
- ☐ 10.2: All other expression categories
- ☐ 11: Undefined identifiers
- ☒ 11.1: Undefined local variables
- ☒ 11.2: Undefined member data
- ☐ 11.3: Undefined functions (free and member)
- ☐ 12: Function calls made with right number and type of parameters
- ☐ 13: Arrays should ...
- ☒ 13.1: Use the same number of dimensions as declared
- ☐ 13.2: Use integer for indexes
- ☐ 13.3: Use compatible dimensionality when passed to function based on declaration

Section 2. Design

The overall design of the semantic analysis relied heavily on the abstract syntax tree developed previously. Unfortunately, due to many issues with the parsing aspects, mainly due to a hot mess of a grammar, a lot of wasted efforts occurred. Eventually, it was decided to simply rewrite the grammar from scratch using clearer nomenclature. This made the design must simpler afterwords, and it was much easier to detect any flaws in the modified grammar. Consequently, after encountering so many issues, amplified by using F# without significant familiarity, led to lack of time and hence an incomplete assignment. Nevertheless, the remaining semantic checks are intended to be implemented for the last assignment.

In approaching the semantic analysis, the chosen strategy was to separate the concern of individual 'checks' as much as possible. This was done in order to well suit the immutable and highly recursive nature of F#. Like all functional languages, immutability and recursion are first class concepts. However, when choosing to not implement data structures to be mutable, this makes it very unfavorable to create and operate on data at the same time. To this end, the symbol table was created individually (i.e. no semantic checks were performed in this initial phase). The `SymbolEntry` structure was implemented as a recursive data structure, i.e. one of its fields is a list of `SymbolEntry` (often aliased as a `SymbolTable`), denoting its direct nested scopes. This follows a similar structure to that of the AST in the sense that each root structure contains a list of the same structure type. Following the symbol table creation phase, a set of various phases were implemented.

The various phases of the semantic analysis were implemented what were chosen to be referred to as 'visitors'. This is simply to emulate the visitor pattern used often in OOP languages. Since F# is functional and supports pattern matching and discriminated unions, implementing functions that are able to exhaustively execute various branching logic against the cases of a discriminated union is very intuitive. The implementation for this can be found all in `Semanter` for the semantic analysis and in `SymbolTable` for the table creation.

Three visitor modules were designed with different goals: `SymbolCheckVisitor`: Traverses the symbol table and determines any errors related to information available with only the symbol table (such as undeclared identifiers, multiple declared identifiers, etc).

`SymbolTableVisitor`: Traverses the AST and 'lifts' any nested types up to its parent if needed. For example, in

`VarElementLists`, its child nodes `DataMember` and `FunctionCall` require to have types, to analyze, which are not determined from symbol table creation only.

`TypeCheckVisitor`: Traverses the AST and determines any errors in typed expressions. Relies on `SymbolTableVisitor` since it requires that all nodes 'lift' their types to parent nodes when necessary. For example, `AssignStat` requires its child nodes to be typed, which only occurs after `SymbolTableVisitor`.

Please note that the test project is where the project can be easily run and output the necessary files. The output files are located in `Moon.Tests/resources/grammar/out/<test-file-name>`. Unfortunately, there was no time to run a proper command line driver. The symbol table files end in `.parse.symbols` and the AST have `.dot.pdf` generated if you have the `dot` tool installed locally on your computer.

Section 3. Use of Tools

No new tools were used in this part of the project that were different from previous assignments.