Symbol Table Design

The symbol table for this assignment (implemented in `src/compiler488/symbol/`) was designed much like the symbol table described in section 8.3.3 of the textbook (Fischer, Cytron, LeBlanc Jr. – *Crafting a Compiler*).

More specifically, there are several parts to our symbol table design, all of which are described below:

`SymbolTable.java`

This class has 3 attributes

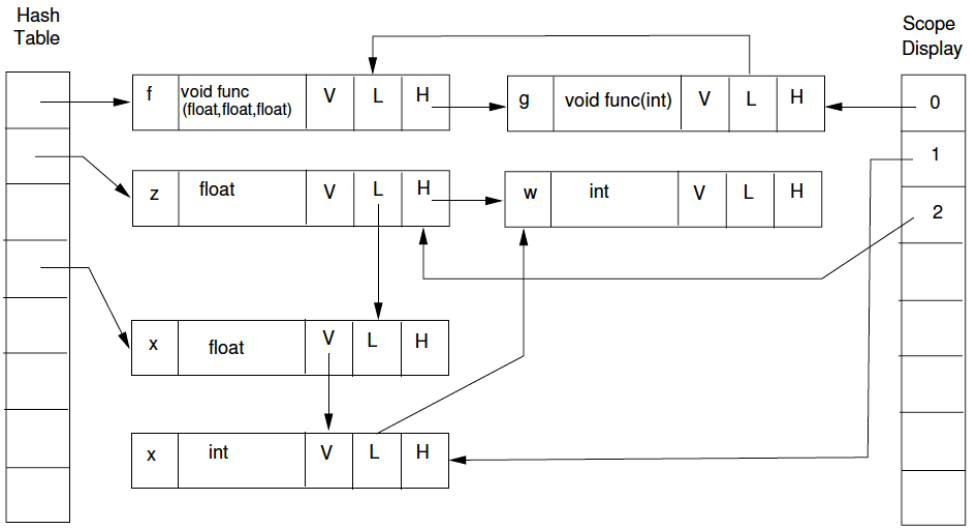
- A hashmap of all symbols that have been declared thus far (in all scopes)
- A list of list of symbols. Each element in the top-level list contains the list of symbols (each of which is a `SymbolTableEntry`) that have been declared at a given scope. The `i`'th element of the top-level list contains the list of symbols declared at scope of depth `i`
- A `depth` variable that tracks the current depth (in terms of scopes and nested scoping) that is currently being semantically analysed

In other words, there is a hashmap that is maintained at all times containing all the variables declared in the program thus far. The hashmap is pivotal for fast lookup of symbols (which is crucial in the compiler). Then, there is also the above-described list of list of symbols which helps us keep track of symbols declared at each scope, and which scope(s) they apply to.

Some member functions in this class are:

- `openScope` – functionality when a new scope is encountered/entered in the program
- `closeScope` – functionality when a scope is exited. All symbols in this scope are destroyed, and are replaced in the hashmap by any previously-declared (in outer scopes) symbols with the same name.
- `getEntry` – lookup and return a symbol with a given name
- `addEntry` – add a newly-encountered/declared symbol in the hashmap and list of lists (at the current depth) as appropriate

Below is a diagrammatic representation of our symbol table design, taken from the section 8.3.3 from the textbook (as our design is mostly the same):



`SymbolTableEntry.java`

Every symbol in the hashmap and list of lists from `SymbolTable` are `SymbolTableEntry` objects.

Here, we have a hierarchy of symbols, since we know that there are various kinds of symbols:

`SymbolTableEntry`

Every symbol is a subclass of this. Every symbol (i) has a name, (ii) can have a previously-declared symbol of the same name (declared in an outer scope), (iii) has a depth (of scoping), and (iv) has a list of other declared symbols in the same scope

- **`VariableSymbol`**
  There are two types of symbols, one of which is a `VariableSymbol` (symbols representing Integers, Booleans, arrays, etc.). In addition to the attributes in its superclass, every `VariableSymbol` also has a `Type`.
  - **`ScalarSymbol`**
    There are two types of `VariableSymbol`s, one of which is a `ScalarSymbol` (Integers and Booleans). This has no additional attributes to its superclasses.
  - **`ArraySymbol`**
    This is the other kind of `VariableSymbol` – this applies to every array symbol. In addition to the attributes of its superclasses, `ArraySymbol`s have lower and upper bound, and a size.
- **`RoutineSymbol`**
  This is the other type of symbol – this applies to functions and procedures. In addition to the attributes of its superclass, it also contains a list of the `Type`s of the parameters of the routine (which will be used in semantic analysis for type-checking).
  - **`ProcedureSymbol`**
    There are two types of `RoutineSymbol`s, of which is a `ProcedureSymbol` (applies to procedures). This has no additional attributes to its superclasses.
  - **`FunctionSymbol`**
    This is the other kind of `RoutineSymbol` – this applies to functions. In addition to the attributes of its superclasses, `FunctionSymbol`s also have a return `Type`.