Design Decisions

Symbol Table

My symbol table is constructed of a "cactus" stack of hashmaps, using the hashing function given in the lecture slides. I elected to use this hashing function because, as shown in the slides, it works well. Each hashmap in the symbol table, or *symbol table frame*, consists of entries of symbols, mapped to by their names (identifiers). A symbol has the following form:

```
typedef struct SYMBOL {
   int lineno;
   char* name;
   SymbolKind kind;
   union {
      struct TYPE* typeS;
      struct TYPE* varS;
      struct FUNCTIONDECLARATION *functionDeclS;
      struct {struct STATEMENT* statement;
            struct TYPE* type; /* will be NULL until type checking */} shortDeclS;
      struct {struct TYPEDECLARATION * typeDecl;
            struct TYPE* type;} typeDeclS;
      struct {struct VARDECLARATION* varDecl;
            struct TYPE* type; /* could be NULL */} varDeclS;
      struct {struct PARAMETER * param;
            struct TYPE* type;} parameterS;
      struct {struct FIELD* field;
            struct TYPE* type;} fieldS;
   } val;
   struct SYMBOL *next;
} SYMBOL;
```

As can be seen, we take care to keep track of types in the symbol table, as this becomes important in the type checking phase to follow. Additionally, on each symbol we store a reference to the AST node corresponding to the declaration of that symbol, as this is also useful to have in the type checking phase. The symbols typeSym and varSym with values val.typeS and val.varS respectively are for pre-declared variables and types.

Each new scope entered pushes a new hashmap onto the stack. We create a new scope:

- For the "universe" (pre-declared types and variables)
- For the outermost scope of the file (top level declarations)
- For each function body
- For each entire if statement, if/else statement, switch statement, and for statement. We need to do this so that the initialization statements can "live" in their own scope.
- For the body of an if statement
- For the body of an else statement
- For the body of a for statement

- For the body of each switch case
- For any new block

Type Checking

Type checking, in essence, revolves around type checking statements and expressions. Statement type checks first involve type checking the expressions composing the statement, and then follow these checks by checking that the statement overall is well-typed, given the types of the expressions. For example, we might first type check the expressions on both sides of an assignment statement and then confirm that the resulting types are the same. Expression type checking ensures that a given expression has the expected type. In this step, we check that expressions are numeric when they need to be, booleans when then need to be, comparable and ordered when they need to be, etc.

In the expression type checking, we finally convert AST nodes saved as function calls but which are actually casts to casts, and then we re-type check the cast.

On a final note, in order to compare identifier types to determine if they are identical, we need to know if they were declared in the same type specification. To do this, we store a unique number on each TYPEDECLARATION node in the AST and then compare these numbers. This seems slightly hack-ish, but given limited time constraints I went with it for now.

Note that I do not have time to write out what each of my test programs does – I am sorry! However, there is a comment in each detailing what the error is.

Finally, note that I did not have time to implement the flag –pptype, nor did I have time to implement type checking of struct field access.