Compiler Project Report

Wayne Stegner

April 20, 2021

Code Structure

My compiler is written in C++ with an object-oriented approach. It contains the scanner, parser, and type checker for now, but I intend to complete code generation and runtime this Summer. The `Parser` class facilitates the overall flow of the recursive-descent parse. It interacts with the `Scanner` class to grab one token at a time, only ever looking at the current token. While the `Scanner` produces the tokens, it does not add them to the symbol table because it lacks semantic information such as scope and whether it should lookup or insert the symbol. Instead, the `Parser` inserts/looks up symbols through the `Environment` at appropriate places in the parse tree.

The `Environment` abstracts scoping from the symbol table except for the case of inserting a symbol, where the `Parser` passes a `bool` value to distinguish global vs local inserts. The `Environment` class holds a global `SymbolTable` object and a local stack of `SymbolTable` objects to accommodate the scoping requirements. The `SymbolTable` class is mostly a wrapper to handle lookups and insertions to a STL hash map with string keys and token values.

Tokens are implemented in multiple types using polymorphism, and they also serve as the symbol table entries. The base `Token` class stores a `TokenType` enumeration to help the parser identify grammar elements, a string denoting the token value, and a `TypeMark` enumeration for data type information. The `IdToken` class is used for identifiers, both variables and procedures. This class inherits from `Token`, adding an integer for counting elements, a boolean for denoting procedures, and a vector of `IdToken`s for holding parameter types. The `LiteralToken` template class is used to allow different data type values to be returned from the `Scanner`. Each token is returned from the scanner as a base `Token` pointer, and they are dynamically cast to child classes as needed in the `Parser`.

The `TypeChecker` class holds functions useful for checking compatibilities of data types and array sizes for various operators. The functions are invoked during the parser, but the functions abstract some semantics regarding type matching (e.g. floats and integers are compatible for addition, but not a bitwise operator). The array size checking ensures that the arrays are either the same size or one of the operands is a scalar value.

For error recovery, the parser takes various actions depending on the error. Type mismatches or size mismatches generally just require reporting the error and then continuing the parse as normal. However, the case of an unexpected token typically results in a panic mode recovery. Recovery mode uses `;` and `EOF` as sync tokens, and when a panic happens in either a `<declaration>` or a `<statement>`, the current production is abandoned. I plan to make error recovery a bit more intelligent as I finish the compiler this Summer.

Running the Compiler

Prerequisite: `make` and `gcc` are installed. Run `make` in the project root directory, producing the executable `./bin/compiler`. Run `./bin/compiler -h` for usage. Example: `./bin/compiler -l 1 -i code.src -l log.txt` compiles `./code.src` with log level info and saves the full debug log to `./log.txt`. No binary is produced (yet).