Compiler Project Report

Wayne Stegner

August 9, 2021

Code Structure

My compiler is written in C++ with an object-oriented approach. It contains the following:

Scanner: The scanner is simple enough. It outputs 1 token at a time when requested from the Parser. It doesn't touch the symbol table because it lacks semantics relative to the Parser.

Parser: The parser is a recursive descent parser, as per the project spec. It is what I would consider the "main" module, and it drives all of the other components of the compiler. Some modifications to the grammar were made to simplify the parse. The declaration lists and statement lists were given their own definitions to make it flow a little better. The `expression` and other operator nodes had left recursion and were broken into two nodes, adding the `Prime` variant of each. The originally named nodes don't do much in the parse; they call the parsing of the left-hand side of the parse tree, then they hand the information to the `Prime` nodes for parsing the right-hand side, and that is where the fun stuff happens (type checking and code generation, namely).

Symbol Table/Environment: The symbol table and scoping rules are encapsulated in the environment, which holds a global symbol table and then a stack of local symbol tables. The symbol tables hold the tokens from the scanner, which hold lots of semantic information used throughout the compilation process. The environment handles variable scoping for the parser during lookups.

Type Checker: The type checker holds some helper functions to make sure types are compatible for the given operators. The parser invokes the type checking functions right before code generation stages which deal with types. Type checking also checks array sizes.

Error Recovery: There are a couple of error recovery strategies. Type checking violations raise an error, but do not impede the flow of the parser, as long as the code is syntactically correct. However, unexpected tokens cause a panic mode, which syncs on `;` and `EOF` tokens. A panic will cause the current production to be dropped, although it doesn't always happen super cleanly. As they say: garbage in, garbage out.

Code Generation: I tried several different code generation methods, but I found generating the `llvm` commands by hand to be the best route for me. (I wanted to use the API but I could not figure out how to install it... `vim` users, am I right?) I made my own module to generate the necessary commands broken into several sections which I merge into 1 string at the end and emit to a file. The basic principle is to store and manage handles to `llvm` objects, which makes it easy to string together a multitude of commands. It's not the most efficient, but that's what optimization passes are for, right?

The biggest code generation challenges were dealing with nested procedures and branch structures. However, these were taken care of with a few well-thought-out stacks. There is a stack to store procedure information, such as the current register count, the code generated so far, and counters for labels. When a new procedure starts in the source code, a fresh procedure struct is pushed to the stack, and then when a procedure ends, the procedure struct is popped and emitted to the main body of code. A similar stack strategy is used for loops and if statements, though these stacks only contain the branch labels.

Runtime Environment: The runtime was made by compiling a c program to a shared object and then loading that when running `lli`. The c program is in the `src/runtime` directory.

Running the Compiler: Dependencies: `clang/llvm` 6.0.0, `make` 4.1, `gcc` 7.5.0. The `llvm` version is old because that's what was on my system and I had dependencies I did not want to break. Some of the syntax and semantics seem different from the version discussed in class, mostly regarding implied register numbering.

Run `make` in the project root directory to compile my compiler and the runtime shared object. Then, generate the `.ll` file with `./bin/compiler -l 2 -i code.src` (assuming `code.src` is the relative path to the source file) (see `./bin/compiler -h for full usage`). The `.ll` file will emit to `bin/`, where you can compile it with `llvm-as bin/code.ll`, then fire up the interpreter with `lli --load=bin/runtime.so bin/code.bc`.

Alternatively, running `make all` should *hopefully* make all of the `.bc` files, then you can run them with the `lli` command above.