Introduction to Compilers and Language Design

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at this website:
`http://compilerbook.org`

Draft version: September 14, 2018

Appendix A – Sample Course Project

This appendix describes a semester-long course project which is the suggested companion to this book. Your instructor may decide to use it as-is, or make some modifications appropriate to the time and place of your class. The overall goal of the project is to build a complete compiler that accepts a high level language as input and produces working assembly code as output. It can naturally be broken down into several stages, each one due at an interval of a few weeks, allowing for 4-6 assignmnets over the course of a semester.

The recommended project is to use the C-Minor language as the source, and X86 or ARM assembly as the output, since both are described in this book. But you could accomplish similar goals with a different source language (like C, Pascal, or Rust) or a different assembly language or intermediate representation (like MIPS, JVM, or LLVM.)

Naturally, the stages are cumulative: the parser cannot work correctly unless the scanner works correctly, so it is important for you to get each one right, before moving onto the next one. A critical development technique is to create a large number (30 or more) test cases for each stage, and provide a script or some other automated means for running them automatically. This will give you confidence that the compiler works across all the different aspects of C-Minor, and that a fix to one problem doesn't break something else.

A.1 Scanner Assignment

Construct a scanner for C-Minor which reads in a source file and produces a listing of each token one by one, annotated with the token kind (identifier, integer, string, etc) and the location in the source. If invalid input is discovered, produce a message, recover from the error, and continue. Create a set of complete tests to exercise all of the tricky corner cases of comments, strings, escape characters, and so forth.

A.2 Parser Assignment

Building on the scanner, construct a parser for C-Minor using Bison (or another appropriate tool) which reads in a source file and determines whether

the grammar is valid, and indicates success or failure. Use the diagnostic features of Bison to evaluate the given grammar for ambiguities and work to resolve problems such as the dangling-else. Create a set of complete tests to exercise all of the tricky corner cases.

A.3 Pretty-Printer Assignment

Next, use the parser to construct the complete AST for the source program. To verify the correctness of the AST, print it back out in as an equivalent source program, but with all of the whitespace arranged nicely so that it is pleasant to read. This will result in some interesting discussions with the instructor about what constitutes an "equivalent" program. A necessary (but no sufficient) requirement is that the output program should be re-parseable by the same tool. This requires attention to some details with comments, strings, and general formatting. Again, create a set of test cases.

A.4 Typechecker Assignment

Next, add methods which walk the AST and perform semantic analysis to determine the correctness of the program. Symbol references must be resolved to definitions, the type of expressions must be inferred, and the compatibility of values in context must be checked. You are probably used to encountering incomprehensible error messages from compilers: this is your opportunity to improve upon the situation. Again, create a set of test cases.

A.5 Optional: Intermediate Representation

Optionally, the project can be extended by adding a pass to convert the AST into an intermediate representation. This could be a custom three or four-tuple code, an internal DAG, or a well established IR such as JVM or LLVM. The advantage of using the later is that output can be easily fed into existing tools and actually executed, which should give you some satisfaction. Again, create a set of test cases.

A.6 Code Generator Assignment

The most exciting step is to finally emit working assembly code. Straightforward code generation is most easily performed on the AST itself, or a DAG derived from the AST in the optional IR assignment, following the procedure in Chapter 11. For the first attempt, it's best not to be concerned about the efficiency of the code, but allow each code block to conservatively stand on its own. It is best to start with some extremely simple programs (e.g. `return 2+2;`) and gradually add complexity bit by bit. Here, your practice in constructing test cases will really pay off, because

you will be able to quickly verify how many test programs are affected by one change to the compiler.

A.7 Optional: Extend the Language

In the final step, you are encouraged to develop your own ideas for extending C-Minor itself with new data types or control structures, to create a new backend targeting a different CPU architecture, or to implement one or more optimizations described in Chapter 12.