Overview of Compiler Implementation Projects

COMPSCI 322 — Computer Languages and Compilers — Spring 2015

February 9, 2015

Overall Structure of the Compiler

The compiler is one unified software system whose development is divided into four parts:

1. Scanner
2. Parser / semantic analyzer
3. Optimizer
4. Code generator

Projects 3 and 4 may have their order reversed. If so, I'll announce it when Project 3 is assigned.

Interfaces between Parts

Note. This section may change during the semester.
We will use a simple interface between the four parts:

1. Scanner reads a Decaf source code file. It must return an ordered list of tokens, in the order they were found.
2. Parser / semantic analyzer uses the scanner's output. It must return:
   (a) A symbol table, annotated with types and values for each symbol; this is generated from list of tokens provided by scanner)
   (b) A parse tree / abstract syntax tree (AST), or other high-level IR (I will define a specific interface / output spec for the high-level IR later)
3. Optimizer uses parser / semantic analyzer's output, must return
   (a) a low-level IR (e.g., three-address code (TAC)) with basic optimizations done
   (b) symbol table with any extra data that needed to be added (maybe none)
4. Code generator uses optimizer's output, must return MIPS code that is ready for QtSpim to run

Requirements for All Parts

- All parts must return appropriate error messages and abort on fatal errors.
- The compiler must maintain all of the intermediate results returned by the first three passes.
- Additionally, each stage of the compiler except for the code generator must provide methods to "pretty-print" all output from that stage, for debugging purposes.
- Specific output formats may be required for some phases of the compiler, particularly projects 1 (scanner) and 4 (code generator); this will make debugging (for you) and grading (for me) easier.

Infrastructure

Your group has the following resources available to it:

1. Individual accounts on the (new!) `washington.uww.edu` server, with access to:
   - text editors: nano, vim, and emacs
   - compilers: gcc/g++ 4.3.4 and javac 1.6.0 (if you want to use Java 1.7/1.8 features, talk to me)
   - debuggers (text-based): gdb (C/C++) and jdb (Java)
   - build managers: make (C/C++/Java) and ant (Java)
   - version control: git (client only, if you want to use a Github repository); I might be able to install subversion if there's interest

   Note. Even if you develop on another system, make sure that your code compiles and runs on `washington`. I will build and test your programs using automated test scripts on `washington`. Programs that do not compile for any reason (other than an incompatible Java version) will earn Fix and Resubmit grades!

2. A group dropbox folder on D2L for each project. Your group will submit a single archive file to the dropbox folder for each project, which **must include source code**. Most common formats (.jar, .zip, .tar.gz/.tgz, .tar.bz2/.tbz2) will work. I can read .rar and .7z archives, but not without complaining.

3. A discussion board for your group on D2L. Anything you post on here is visible only to your group and to me. I'll monitor these discussion boards for questions where I can give helpful advice. If there's interest from the class, I can create a class-wide discussion board as well.

Version Control

Using version control is *strongly encouraged*, but not required, for projects in this course. You and 2-3 other people will be working on fairly complex code at the same time. If someone in your group makes changes that break your program, version control could save you hours (or days) of work.

- If your group members have Github accounts, then you may have a version-control solution already. Use the `git` command on washington to do standard git-related tasks from the command line.
- If your group wants to use Subversion instead, talk to me so I can (maybe) set up a SVN repository for your group on washington. (It may take a few days.)

Build Manager

Using a build manager is not required (and won't be covered in class), but might be helpful. If you have many files in your project, writing a script for a build manager may make building your project easier. The `ant` and GNU `make` build managers are available on `washington`.

- `make` is the "standard" build manager on Unix-like (POSIX) systems. You can write a Makefile for any languages or compilers, but `make` is usually used for C/C++ projects. Manual: `info make` on `washington`, or `https://www.gnu.org/software/make/manual/` on the WWW.
- `ant` is a commonly used build manager for Java projects. Manual: `http://ant.apache.org/manual/index.html`.
- Tutorials for both tools are available online. Use your favorite search engine to find them.

If you use a build manager, mention this in your project report (see next section) and include the Makefile or `build.xml` file in your project archive file. **Test your build script on** `washington` **before submitting!**

Project Reports

A written report must be included in your project archive when you submit it.

Format and Style

Your report should be written in standard English, though it does not need to be as formal as a research paper (e.g., feel free to use bulleted lists). It should be clear, concise and readable. There is no specific length requirement: it should be long enough to tell me the information you think I should know, but no longer.

Plain text is perfectly acceptable; fancy formatting is not necessary (but consider "faking" bulleted lists and section headings if you use plain text, for easier reading). If you use a fancier format, submit a .pdf file or any format that Microsoft Word can open (e.g., .doc[x], .odt, .rtf).

Required Parts

Your documentation must include the following parts. (*Hint*: Make each of these parts a separate section.)

1. How to build your project from the source code. If you use `make` or `ant`, tell me. If not, you must include the full command(s) you used to build your compiler. (Remember, if I can't figure out how to compile your project, your group gets a Fix & Resubmit grade for the project.)

2. A brief description of how your group divided the work. Summarize each group member's contribution to the project.

3. A list of any clarifications, assumptions, or additions to the problem assigned. The project specifications are fairly broad and leave many of the decisions to you. This is an aspect of real software engineering. If you think major clarifications are necessary, please ask the instructor.

4. An overview of your design, an analysis of design alternatives you considered (if any), and key design decisions. Be sure to document and justify all design decisions you make. Any decision accompanied by a convincing argument will be accepted. If you realize there are flaws or deficiencies in your design late in the implementation process, discuss those flaws and how you would have done things differently. Also include any changes you made to previous parts and why they were necessary.

5. A brief description of interesting implementation issues. This should include any non-trivial algorithms, techniques, and data structures. It should also include any insights you discovered during this phase of the project. (External sources of implementation help, e.g., websites or books, should be cited here.)

6. A list of known problems with your project, and as much as you know about the cause.

   - If your project fails a provided test case, but you are unable to fix the problem, describe your understanding of the problem.
   - If you discover problems in your project in your own testing that you are unable to fix, but are not exposed by the provided test cases, describe the problem as specifically as possible and as much as you can about its cause. If this causes your project to fail hidden test cases, you may still be able to receive some credit for considering the problem. If this problem is not revealed by the hidden test cases, then you will not be penalized for it.

The write-up for each project is worth about 25% of the project grade. Make sure to work on it in parallel with your program development. Don't wait until a few minutes before the deadline to start it!

Acknowledgements

The compiler implementation projects for this course are adapted from the projects developed by Profs. Saman Amarasinghe and Martin Rinard for the Spring 2010 offering of 6.035 Computer Language Engineering at the Massachusetts Institute of Technology (MIT). In particular, the project report requirements are nearly the same as Amarasinghe and Rinard required for their students, because I think they made it into a thought-provoking assignment — and what good is a university education if it doesn't teach you to think?

The original course materials for 6.035 Computer Language Engineering are available from the MIT OpenCourseWare website at `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/projects`. They are adapted for this course by the permissions granted by the Creative Commons BY-NC-SA 4.0 license under which they are made available. To comply with the terms of this license, I will mirror copies of the project assignments on my public website for the course (`http://cs.uww.edu/~osterz/322/`) as they are posted.