15-411 Compiler Design: Lab 6 - LLVM
Fall 2009

Instructor: Frank Pfenning
TAs: Ruy Ley-Wild and Miguel Silva

Compilers due: 11:59pm, Sunday, December 3, 2009
Term Paper due: 11:59pm, Thursday, December 10, 2009

1 Introduction

The main goal of the lab is to explore advanced aspects of compilation. This writeup describes the option of retargeting the compiler to generate LLVM code; other writeups detail the option of implementing garbage collection or optimizing the generated code. The language *L4* does not change for this lab and remains the same as in Labs 4 and 5.

2 Requirements

You are required to hand in two separate items: (1) the working compiler and runtime system, and (2) a term paper describing and critically evaluating your project.

3 Tests

You are not required to hand in new tests. The autograder will use a subset of the tests from the previous labs to test your compiler.

4 Compilers

Your compilers should treat the language *L4* as in Labs 4 and 5, including `extern` declarations. While we encourage you to continue to support both safe and unsafe compilation, but it complies with the specification if the potentially unsafe implementation is simply the safe one.

When generating code for the LLVM, given file *name*`.l4`, you compiler should generate: *name*`.ll`, which is in the LLVM human-readable assembly language. The driver will use LLVM commands to generate from this the file *name*`.s`, in the x86-64 assembly language. Translation from the former to the latter is in at least two stages, first generating an intermediate byte code file *name*`.bc` with `llvm-as` and then *name*`.s` by using `llc` (see the LLVM documentation).

If you would like to apply optimizations yourself, your compiler may directly generate the *name*`.bc` file, *in addition* to the *name*`.ll` file, in which case the script will ignore the *name*`.ll` file. However, you must still generate it for the purpose of grading your compiler.

The LLVM compilation tools can be found in

`/afs/cs.cmu.edu/user/fp/courses/15411-f09/llvm/bin`

This directory is not in Autolab's path, so your script need to add that or directly address the executable you want to run (as does the grading script).

5 Project Requirements

On the Autolab server, the hand-in and status pages for the optimization and garbage collection projects are separated, since different drivers will be employed.

Compiler Files (due 11:59pm on Thu Dec 3)

As for all labs, the files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a roadmap to your code. This will be a helpful guide for the grader.

Issuing the shell command

```
% make l4c
```

should generate the appropriate files so that

```
% bin/l4c --safe --llvm <args>
% bin/l4c --unsafe --llvm <args>
% bin/l4c --safe --x86_64 <args>
% bin/l4c --unsafe --x86_64 <args>
```

will run your *L4* compiler in safe and unsafe modes, generating LLVM or direct x86-64 native code, respectively. For backwards compatibility, the default is `--unsafe --x86_64`. The driver will only use the `--llvm` flag to autograde your code.

In order for you to be able to provide a runtime system or library functions, the compiler expects a file `l4lib.c` at the top-level of your compiler directory and compile and link this against your file when compiled in `-llvm` mode.

You may decide just to support either safe or unsafe modes, which should then become the default. The compiler should exit with an error code if given an unsupported switch.

The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

Using the Subversion Repository

The handout files for this course can be checked out from our subversion repository via

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>
```

where `<team>` is the name of your team. You will find materials for this lab in the `lab6llvm` subdirectory. Or, if you have checked out `15411-f09/<team>` directory before, you can issue the command `svn update` in that directory.

After first adding (with `svn add` or `svn copy` from a previous lab) and committing your handin directory (with `svn commit`) to the repository you can hand in your tests or compiler by selecting

```
S3 - Autograde your code in svn repository
```

from the Autolab server menu. It will perform

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>/lab6llvm/compiler
```

to obtain the files directories to autograde, depending on whether you are handing in your test files or your compiler.

If you are submitting multiple versions, please remember to commit your changes to the repository before asking the Autolab server to grade them! And please do not include an compiled files or binaries in the repository!

Term Paper (due 11:59 on Thu Dec 10)

You need to describe your implemented compiler and critically evaluate it in a term paper of about 10 pages. You may use more space if you need it. The recommended outline varies depending on your project. Submit a file `<team>-llvm.pdf` via email to the instructor at `fp@cs`.

Your paper should follow this outline.

1. Introduction. This should provide an overview of your implementation and briefly summarize the results you obtained.

2. Comparison. Compare compilation to LLVM, followed by native code generate with direct native code generation. How does the structure of your compiler differ? How does the generated code differ? If you are applying optimizations at the LLVM level, describes those optimizations and their rationale.

3. Analysis. Critically evaluate the results of your compiler via LLVM, which could include size and speed of the generated code. You might find the driver for the optimization lab `lab6opt` to be useful for this purpuse. Also provide an evaluation of LLVM: how well did it serve your purpose? What might be improved?

6 Notes and Hints

- Apply regression testing. It is very easy to get caught up in writing a back end for a new target. Please make sure your native code compiler continues to work correctly!

- Read the assembly code. Just looking at the assembly code that your compiler produces will give you useful insights into what you may need to change.

- The intermediate form accepted by LLVM must be in SSA form. However, it is possible to allocate all variables on the stack and use a script provided with LLVM in order to convert into SSA form. See Chapter 4.7 of the LLVM Tutorial.

- The LLVM, like C, leaves the result for certain operations undefined (e.g., division by 0), so you may need to turn them into function calls.

- Additional hints may be found under LLVM Notes at the course home page.