15-411 Compiler Design: Lab 5
Fall 2007

Instructor: Frank Pfenning
TAs: David McWherter and Noam Zeilberger

Benchmarks due: 11:59pm, Thursday, November 29, 2007
Compilers due: 11:59pm, Thursday, December 6, 2007
Term Paper due: 11:59pm, Thursday, December 13, 2007

1 Introduction

The main goal of the lab is to explore advanced aspects of compilation. You have the choice of two projects: implementation of various optimizations for *L3* or implementation of a garbage collector for *L3*.

2 Requirements

You are required to hand in three separate items: (1) benchmarks which can be used to evaluate your optimizations or test files for your garbage collector, (2) the working compiler and runtime system, and (3) a term paper describing and critically evaluating your project. The language *L3* remains unchanged from Lab 4.

3 Benchmarks and Tests

Optimization Benchmarks

If you are planning to implement optimizations, you should submit benchmarks. The benchmark files you submit must be as in previous labs except that they should also define several specific functions that will be used by the driver for timing purposes. Your compiler should not verify the presence of these functions so we can still run it on older suites for the purpose of regression testing. Your functions should work in both safe and unsafe modes.

- τ`* init(param : int);` This function should create an instance of your benchmark problem, such as an array to sort. The size of your data structure and therefore the running time of your benchmark should vary with `param`. The `init` function will not be timed. The type τ may be different for different benchmarks, since the driver only passes this pointer around without examining its value.

  We encourage you to use the library function `int rand();` to generate random elements in your data structure each time `init` is called. The result of `init` should be predictable and portable, depending only on the random seed which the driver (but *not* your `init` function) can set by calling the C library function `srand`.

- `int prepare(data : ` τ`*, param : int);` This function should prepare an instance of your problem for a timed run, but the time for this function itself will not be counted. For example, you might copy a given array so it can be sorted in place by the `run` function. The return value will be ignored.
- `int run(data : ` τ`*, param : int);` This function should run your algorithm on the data structure and should modify it in some way to record the result. The time for this function call is the timing data we collect. This function should not call external library functions: we want to test the efficiency of your code generator, not that of the library functions. The return value will be ignored.
- `int checksum(data : ` τ`*, param : int);` This function should reduce your output to a checksum. The driver will use this to compare the checksum with the checksum obtained from a reference implementation in order to catch errors. You should try to write this function so it is likely to catch errors. In simple cases, such as function that computes an integer, the checksum can be the value itself.

The driver will call `init` once to get a (deterministic) problem instance. It will then call `prepare` followed by `run` and continue to repeat the `prepare`–`run` sequence a fixed number of times, calculating an average running time for `run`. After the last run, `checksum` is called and compared with the checksum of the reference implementation. It is important that `prepare` followed by `run` is idempotent, so that each successive run should execute exactly the same way.

The driver will call `init (1000)`; the run function for this parameter should take about 100ms with the reference implementation, but you do not need to fine-tune this.

For ease of regression testing, your `main` function should, deterministically, call `init`, `prepare`, `run`, and then return the result of `checksum`. The first line of the file should read `#test return` n where n is that (deterministic) checksum.

A general strategy is to turn interesting test files from earlier labs into benchmarks.

Garbage Collector Test Files

If you are planning to implement a garbage collector, you should submit garbage collector test files. We will not test the garbage collector for efficiency, only for correctness.

The format of the test files is as for Lab 4, except that you should add a function `int test(param : int)`. The memory usage of the program should go up with the parameter. We encourage you to use the library function `int rand();` to generate random test instances. The result of `test` should be predictable and portable, depending only on the random seed which the driver can set by calling the C library function `srand`.

You should also supply a function `main` which calls `test` after seeding the random number generator with a fixed number. The output of this function should be the expected return value in the first line.

The driver will call `test` multiple times with different parameters, random seeds, and memory limits to verify that the program does not crash. It will also call `main` to verify the correctness of the computation

4 Compilers

Your compilers should treat the language *L3* as in Lab 4, including `extern` declarations. While we encourage you to continue to support both safe and unsafe compilation, you may commit to one or the other compiler and terminate with exit status 1 if `l3c` is called with the unsupported switch.

Optimizations

If you are implementing optimization for your *L3* compiler, you have complete freedom which ones to choose. The ones discussed in lecture so far and the textbook should be considered generally important and constitute good choices. If you would like to specifically target safe compilation you may pick array bounds check elimination which will be discussed in Lecture 25 and you can find in Chapter 18.4 of the textbook.

Grading criteria include first the correctness of the compiler (including the optimizations) and second the scope of the implemented optimization(s). Generally, more widely applicable optimizations should be prefered. However, it is also acceptable to go for a particular complex optimization such as array bounds elimination. As a third criterion, the code quality of the optimization in terms of algorithm, readability, and modularity are considered. Finally, the efficiency of the compiler itself (as compared to the compiled code) is important only to the extent that each benchmarks should compile within a generous time limit of 40 seconds.[1]

Garbage Collection

You have complete freedom which kind of garbage collector to implement. A garbage collector will consist of the compiler proper and the runtime system. The interface from the compiled code to the runtime system should be part of your design. Reasonable choices are a mark-and-sweep or a copying collector, but even a conservative collector is acceptable. Incremental collectors are significantly harder and should only be attempted if you already have a basic collector working. Similarly, you should avoid optimizations; efficiency is only a minor concern for this lab. Functional correctness is paramount, and the absence of memory leaks is a secondary criterion.

5 What to Turn In

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.

Benchmarks or Tests (due 11:59 on Thu Nov 29)

All benchmark or garbage collection test files should be collected into a directory `test/` (containing no other files) and bundled as a tar file `tests.tar` with

```
  % tar -cvf tests.tar tests/
```

to be submitted via the Autolab server. All together, you should submit at least 10 benchmarks or tests. We enourage you to split them evenly in two categories: artificial and realistic. Artificial benchmarks test a particular kind of optimization with a program that is likely to benefit from it. Realistic benchmarks are programs that compute something intuitively meaningful, for example, sorting or matrix multiplication. A similar distinction applies to the garbage collector, where an artificial test just allocates memory according to a simple pattern where some of it may become garbage, and realistic tests employ standard algorithms on data structures such as trees, linked lists, or arrays. You are welcome to submit more than 10 test files.

[1]We reserve the right to revise this upwards based on preliminary handins.

Compiler Files (due 11:59pm on Thu Dec 6)

As for all labs, the files comprising the compiler itself and the runtime system (in case of a garbage collector) 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 l3c
```

should generate the appropriate files so that

```
% bin/l3c --safe <args>
% bin/l3c --unsafe <args>
```

will run your *L3* compiler in safe and unsafe modes, respectively, although as noted above you may choose to support only one of these flags. For backwards compatibility, the default is `--unsafe`.

The command

```
% make clean
```

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

In order to keep the files you hand in to a reasonable size, please clean up the directory and then bundle it as a tar file `compiler.tar`, for example with

```
% cd compiler
% make clean
% cd ..
% tar -cvf compiler.tar --exclude .svn compiler/
```

to be submitted via the Autolab server. If you are using `cvs` instead of subversion, you may want to use the switch `--exclude CVS` instead of `--exclude .svn` for a cleaner hand-in.

Term Paper (due 11:59 on Thu Dec 13)

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 `paper.pdf` on Autolab.

Optimizations

Your paper should follow this outline.

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

2. Optimizations. With a subsection for each separate pass or optimization you implemented, describe what you implemented and briefly state the rationale for your choice. Also give a brief description of the analysis algorithm itself, especially where it deviates from the description in the texbook or other reference material, and how it fits into the overall structure of your compiler.

3. Analysis. Critically evaluate the results of your optimizations. Show graphs or tables that demonstrate the speed-ups or slow-downs you obtained. This requires that you save either performance numbers or implementations (and ideally both) throughout the evolution of your compiler. Explain the results to the extent you can.

You should hand in your paper as a PDF file on the Autolab server.

Garbage Collector

Your paper should follow this outline.

1. Introduction. This should provide an overview of your implementation and briefly summarize the results you obtained.
2. Compilation. Describe the data structures, code, and information generated by the compiler in order to support the garbage collector.
3. Runtime System. Describe the runtime system of the garbage collector, giving details of the algorithms and also its implementation (most likely in C).
4. Analysis. Critically evaluate your collector and sketch future improvements one might make to its basic design.

6 Notes and Hints

- Start small. If you optimize, make sure your instruction selection and register allocation are in decent shape. Improving these is definitely a form of optimization and should be documented in your term paper. If you are building a garbage collector you may want to stay away from optimizations altogether to simplify the interface to the collector as much as possible.
- Apply regression testing. It is very easy to get caught up in the race to faster code. Besides the benchmarks we have a large test suite collected over several labs; use these for regression testing to make sure your compiler remains correct. The same applies to a garbage collector.
- Checkpoint frequently. A convincing term paper should compare before and after for your optimizations, as well as compare to the reference implementation. In order to do this you need to be able to run various versions of the compiler and collect statistics, so make sure you can continue to run older versions. Hand in frequently. Also, it is quite possible you may not be able to finish that last, grand optimization; having a decent prior hand-in is good insurance.
- 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 optimize. You can also run `gcc` on the C code produced by the reference compiler for comparison.

7 Changes

If changes are made to the above specification, they will be summarized here.