Project 3

CMCS 22620/32620, Spring 2004

Assigned: May 5, 2004
Due: May 12, May 19, May 26, June 2 (all 2004)

1 Introduction

This is the final project. It consists of four separate *milestones*. You have one week for each milestone. At the end of milestone 2 you should have a compiler that compiles simple test examples into correctly working executables. The remaining two milestones will enhance the compiler to deal with register spills and also add a simple optimizer.

2 Getting the compiler to work — Milestones 1 & 2

The first two milestones are:

- liveness analysis and interference graph construction (file `liveness.sml`)
- graph coloring (file `color.sml`)

Which of these tasks you tackle first is up to you (although I recommend doing liveness first). Your solution to the first part (as chosen by you) is due on May 12, the second on May 19. At this point your compiler should be able to correctly compile the examples given in the `test/` directory.

2.1 The runtime system

I provided a simple implementation of a runtime system for Minijava in `rt/mj-rt.c`. You need to compile this file using the C compiler:

```
cd rt
cc -c -O2 mj-rt.c
cd ..
```

The Minijava compiler must be invoked from the directory that contains the SML source code, i.e., the same directory where you run the SML compiler. The reason for this is that it looks for a file `rt/mj-rt.o`. (You can change this by editing `main.sml` appropriately.)

3 Cleaning up — Milestones 3 & 4

The remaining two milestones can be done in any order you choose. They will be due on May 26 and June 2, respectively. Here are the tasks:

- Rewrite the instruction stream to deal with spills (file `rewrite.sml`). This involves adding **load**- and **store**-instructions before and after instructions which use temporaries that got spilled, as well as the introduction of new temporaries.
- Implement a simple optimizer for the tree language. This optimizer is supposed to walk over a `LinTree.stm` list and optimize the `LinTree.exp` trees contained therein (file `simplify-tree.sml`). You should implement simple constant-folding and algebraic simplifications such as:

$$\begin{aligned} x + 0 &= x \\ (x + c_1) + c_2 &= x + (c_1 + c_2) \\ x \times 0 &= 0 \\ x \times 1 &= x \end{aligned}$$

  Take commutativity- and associativity-rules into account. (But be sure not to wind up with an infinite loop in the optimizer!)

  (This is an open-ended project. To give you an idea of how much effort I am looking for, note that my sample implementation is 129 lines in total. It handles one case on roughly 2 lines of source code, and in addition to rewriting `LinTree.exp` values it also deals with `CJUMP`s whose conditions are constant.)

4 Instructions

4.1 Files

Download the file `mj-project3.tgz` from the course web page. This is a compressed tarball containing the Minijava compiler's entire source code, with 5 files (`cg.sml`, `liveness.sml`, `color.sml`, `rewrite.sml`, and `simplify-tree.sml`) reduced to mere skeletons to be filled in by you. You should first substitute your solution to Project 2 for file `cg.sml` and then start working on the remaining four.

Notice that I have again improved the frontend of the compiler to the point that it now also handles **static methods**. (With this in place, the `main` method is no longer a special hack but just another static method that is public, returns nothing, and takes an array of String as its argument.)

4.2 SML/NJ

To get started, the first step is to see whether you have access to a working SML/NJ installation. After downloading, uncompressing, and untaring said tarball, you should

end up with a directory named `minijava` containing the source files. Go to that directory and fire up SML/NJ by typing `sml` at the shell prompt. You should see a greeting from SML/NJ and a new input prompt. At this prompt, type `CM.make "minijava.cm";`. The program should compile.

As you make your modifications, you can re-issue the `CM.make` command (without quitting the `sml` session in between). The SML/NJ compilation manager will take care of recompiling only what's necessary.

4.3 The test driver

Among other things the tarball contains two files, `compile.sml` and `main.sml`, which implement a test driver for the Minijava compiler. This lets you build a standalone version and invoke it in a way reminiscent of invoking a C compiler.

The driver understands the following options:

-S Compile to assembly code (`.s`-file) and stop. Neither assembler nor linker will be invoked.

-c Compile to object code (`.o`-file) and stop. The linker will not be invoked.

-o *name* Arrange for the executable to be written to file *name*. If the source file is *foo*`.mj`, then the default for this is *foo*.

Notice that if neither `-S` nor `-c` are given, the Minijava compiler must have access to a file `rt/mj-rt.o` containing the Minijava runtime system.

To build the standalone version of the Minijava compiler, run the following command from your shell prompt:

```
ml-build minijava.cm Main.main mjc
```

This will create a "heap image" named `mjc.ppc-darwin`. The compiler can then be invoked using

```
sml @SMLload=mjc my-program.mj
```

5 Handing it in

To hand in your solution(s), send the file you modified as an e-mail attachment to both the instructor and the TA using the following e-mail addresses:

instructor	`blume (at) tti (hyphen) c (dot) org`
TA	`cysong (at) cs (dot) uchicago (dot) edu`

If you make changes to more than one file, then bundle all your files as a tarball and attach that to your e-mail.

6 Supporting code

There are a number of new source files which I provided for you to use. In particular, you need to familiarize yourself with the implementation of a general graph data structure in `graph.sig` and `graph.sml`. Flowgraphs are described in `flowgraph.sml`. The code for constructing the flowgraph from the list of instruction is in `makegraph.sml`. (See the textbook for an explanations of these.) The register allocator is implemented in `ra.sml`. It invokes the graph coloring phase and the spill rewrite phase.