This week we’ll deal with a full compiler that unites last week’s IR code generator with a register allocator and IR-to-SPARC code generator. Your task will be to improve the quality of register allocation across basic blocks. The assignment must be done on a Sparc.

On the course web page, you’ll find C code for a compiler from IR to Sparc code ircompile.[hc], together with the revised versions of the files from last week, including the IR (ir.[hc]), a translator from byte-code to the IR (irtranslate.c), and a simple optimizer on the IR (iropt.c). (Note that we will no longer need the IR interpreter.) These are connected by a simple main program irmain.c that reads a class, translates it to optimized IR, converts the IR to Sparc code, and executes the result. Once again, the code also relies on David Hanson’s library from C Interfaces and Implementations (CII). There is a makefile to put everything together. Make sure to download the latest version of these files before beginning work, and be on the lookout for possible bug fixes during the week, which will be publicized on the class mailing list.

The Java subset supported is the same as in Homework 4, except that there are some implicit restrictions on the number of simultaneously live variables (since spilling is not supported and in some cases only a limited number of variables can be live across blocks).

As distributed, the system produces fairly decent code (although it could certainly benefit from adding constant propagation and static arithmetic simplification), but it often generates excessive numbers of register-register moves to implement $\phi$-functions. The main source of this problem is that, given a $\phi$-function like

$$ r = \phi(r_1, r_2, \ldots, r_n) $$

there is no mechanism to try to make virtual registers $r, r_1, r_2, \ldots, r_n$ all live in the same physical register. Your main assignment is to fix this problem by coalescing the ranges of these virtual registers into a single range, where possible, and allocating it to a single physical register.

In addition, the current compiler doesn’t cope very well with situations where the sources and destinations of $\phi$’s overlap. Consider the following block entry:

$$ r_1 = \phi(r_2, r_3) \\
 r_2 = \phi(r_1, r_4) $$

Remember that the semantics of $\phi$ functions says that these assignments must be done as if in parallel. It is impossible to coalesce $r_1$ and $r_2$ in this case; in fact, it is also impossible to give the effect of performing the two moves in parallel without using an auxiliary scratch register (or, more baroquely, performing a swap using three $\text{xor}$ instructions). The current compiler is very heavy-handed about this: whenever the sets of sources and destinations overlap, it copies all the sources to scratch registers, and then copies all these to the destinations. Yet in fact, it is always possible to pick an order of copies that has the same effect as a parallel copy, but uses at most one scratch register (and at worst $\lceil n/2 \rceil$ extra copies, where $n$ is the number of $\phi$’ed registers). For additional credit, you can implement such an algorithm.

**Coalescing Details**

In the distributed code, there is a one-to-one mapping from (virtual) registers to ranges. (Hence, allocating a physical register to a range also allocates it to a particular virtual register.) The basic
idea behind coalescing is to change the mapping to be many-to-one by unioning the ranges of the coalesced registers.

To support this, the easiest thing to do is to add a further field to the Regdesc structure containing the representative register for each register. Initially, each register is its own representative. When two registers are coalesced, one of them (chosen arbitrarily) takes the other as its representative. When looking at the register table to determine what physical register (if any) has been allocated to a virtual register, we must now indirect though the representative (possibly multiple times).

Coalescing consists of the following phases:

- Determine sorted live range table for the original registers in the existing way.
- Choose what registers to coalesce. Normally we choose registers that are source and destination of a move (either to implement at \( \phi \) function or as an ordinary OpMove). However, only compatible registers (defined below) can be coalesced.
- Perform the coalescing by choosing one register to be the representative, resetting its range to be the union of the two original ranges, and setting the range of the non-representative to be empty.
- Resort the range table and perform register allocation in the usual way, but taking care to follow representative indirections.
- Generate code in the usual way, but again taking care to follow representative indirections.

Two registers \( r_1 \) and \( r_2 \) are compatible if their ranges are disjoint and

1. neither is pinned; or
2. both \( r_1 \) and \( r_2 \) are pinned to the same physical register \( p \), and no other range that overlaps the union of the ranges of \( r_1 \) and \( r_2 \) is pinned to \( p \); or
3. \( r_1 \) is pinned to physical register \( p \), \( r_2 \) is not pinned, and no other range that overlaps the union of the ranges of \( r_1 \) and \( r_2 \) is pinned to \( p \).

**Homework Credit**

Implementing coalescing is worth 75% of possible points. Alternative approaches to reducing register-register moves at \( \phi \)'s (and elsewhere) are also acceptable. Implementing an algorithm that performs a minimum number of moves at \( \phi \)'s is worth 25%.

**How to submit your homework**

Submit the homework by email to cs577apt@cs.pdx.edu prior to the beginning of class on the due date. You should submit a revised version of file iropt.c, together with any other changed files, as attachments to your mail.