1 Introduction

You should add a code optimization pass to your compiler from assignment 5. The optimizer should improve the code at the intermediate and/or machine code level.

You may want to organize your assignment as a “peephole optimizer”. It does the following:

1. Examine a “window” of instructions, maybe 3-4 instructions long.
2. Improve the code in the window.
3. Slide window down one instruction
4. Repeat until no more changes can be made.

- You get 20 points for implementing the basic optimization framework, and an additional 20 points for every optimizing transformation you implement, up to a total of 100 points.
- For every optimization you implement you must provide one or more (small) test programs. These should be named appropriately (for example, Jump2JumpElim1.gus, FoldConstants3.gus, etc.) and should contain comments that show the grader what the test does and how the optimizer handles it.
- You should add an option -O to your script from assignment 5 to allow optimization to be turned on or off:

```
> lc x.gus > unopt_x.s
> lc x.gus -O > opt_x.s
```

The -O flag should be the last one on the command line.

- If you feel you’ve done something very clever I might be convinced to give extra credit. Be sure to indicate what you have implemented in your README file.
- This assignment can be coded in the language of your choice as long as it can be compiled and run on lectura.

2 Optimizing Transformations

Below are some transformations you might choose to implement. This is not an exhaustive list. You may invent your own (be sure to discuss this with the TA and describe the transformation in the README file) or look in your text book for other ideas.
Redundant loads (20 points): A naive code generator will often generate the same address or variable value several times. The statement \( A := A + 1 \) might produce the SPARC code

\begin{verbatim}
set A, %10
set A, %11
ld [%11], %11
add %11, 1, %11
st %11, [%10]
\end{verbatim}

which could be simplified into

\begin{verbatim}
set A, %10
ld [%10], %11
add %11, 1, %11
st %11, [%10]
\end{verbatim}

Many more simple transformations may be possible.

Jump-to-jump elimination (20 points): Complicated boolean expressions (with many and, or, nots) can easily produce lots of jumps to jumps which can be eliminated:

\[
\text{if } a < b \text{ goto } L1 \Rightarrow \text{if } a < b \text{ goto } L3
\]

\[
\ldots
\]

\[
L1: \text{goto } L2 \Rightarrow L1: \text{goto } L3
\]

\[
\ldots
\]

\[
L2: \text{goto } L3 \Rightarrow L2: \text{goto } L3
\]

Test reversal (20 points): The front-end generates many unnecessary jumps which can be removed by switching the branch direction:

\[
\text{if } a < b \text{ goto } L1 \Rightarrow \text{if } a \geq b \text{ goto } L2
\]

\[
goto L2 \Rightarrow \ldots
\]

You obviously have to be very careful to get the transformation right and avoid doing it when the result would be incorrect. (That is, the optimization needs to be conservative).

Algebraic simplifications (20 points): These are some possibilities:

\[
x := x + 0; \quad \Rightarrow
\]
\[
x := x - 0; \quad \Rightarrow
\]
\[
x := x + 1; \quad \Rightarrow
\]
\[
x := 1 + 1; \quad \Rightarrow \quad x := 1
\]
\[
x := x / 1; \quad \Rightarrow
\]
\[
x := x ** 2; \quad \Rightarrow \quad x := x * x;
\]
\[
f := f / 2.0; \quad \Rightarrow \quad f := f * 0.5;
\]

Constant Folding (20 points): Replace constant subexpressions by their value. Note that many such constant expressions arise from address calculations (such as computing the address of \( x.f.r[9].k \)) and is best done at the machine code level.
Reduction in strength (20 points): Multiplications (and divisions) by constants can be replaced by cheaper sequences of shifts and adds:

\[
\begin{align*}
  x &:= x \times 32 \Rightarrow x := \text{SHL}(x, 5); \\
  x &:= x \times 100 \\
  &\downarrow \\
  x &:= x \times (64 + 32 + 4) \\
  &\downarrow \\
  x &:= x \times 64 + x \times 32 + x \times 4 \\
  &\downarrow \\
  x &:= \text{SHL}(x, 6) + \text{SHL}(x, 5) + \text{SHL}(x, 2)
\end{align*}
\]

3 Extension 1 [20 points]

Write a real program in Luca. By “real” we mean a program that is non-trivial and does something cool. I define what is real and non-trivial; if in doubt talk to me or the TA. (Hint: These programs will, of course, be added to our set of test cases. It is therefore in your best interest to write a really nasty program that your compiler can handle but the other team’s compilers cannot. <=;

4 Submission and Assessment

The deadline for this assignment is midnight, December 11. You should submit the assignment electronically using the Unix command `turnin cs453.6 <files> README ...

Your submission must contain a README-file that states which types of optimizations you can handle. Also, list the name of your team, the team members, and how much each team member contributed to the assignment.

Your electronic submission must contain a working Makefile, and all the files necessary to build the code generator. If your program does not compile “out of the box you will receive zero (0) points. The grader will not try to debug your program or your makefile for you!

This assignment is worth 100 points.

Don’t show your code to anyone outside your team, don’t read anyone else’s code, don’t discuss the details of your code with anyone. If you need help with the assignment see the TA or the instructor.