MC97 Lab 4: Simple Optimizations

Due April 12, 1996; Max Hailperin

In addition to the new modifications to your compiler, you should address any grading remarks from the previous lab that are still relevant.

Next, it is time to implement some simple optimizations (depessimizations might be a better name). We will do this using semantic attributes. Notice that up until now, the only grammar symbols with attributes have been the tokens for identifiers and numbers (and any marker non-terminals involved in control flow). Now we will use attributes for non-terminals as well to implement the simple optimizations addressed by homework 2. For example, consider the assignment statement `a=b+c+(3+4);`. Suppose `a`, `b`, and `c` have been assigned to registers `$s0`, `$s1`, and `$s2` respectively, with the temporaries allocated upward from `$t0`. The simple-minded code we have generated for this until now is:

```
move $t0, $s1
move $t1, $s2
add $t0, $t0, $t1
li $t1,3
li $t2,4
add $t1, $t1, $t2
add $t0, $t0, $t1
move $s0, $t0
```

Now we will start instead generating

```
add $t0, $s1, $s2
add $s0, $t0, 7
```

for a factor-of-four improvement in speed. We will however generally avoid exploiting algebraic properties of the operators. This is for two reasons:

1. It can lead to trouble with issues such as overflow and floating-point roundoff. For example, if `a` and `b` are large number of opposite sign and `c` is also large and of the same sign as `b`, then `a+b+c` interpreted (in the standard way) as `(a+b)+c` might not overflow while interpreted as `a+(b+c)` it would.

2. It turns out to require more sophisticated implementation techniques to fully exploit such algebraic knowledge.

Thus if the parentheses were omitted in the above example, we'd generate

```
add $t0, $s1, $s2
add $t0, $t0, 3
add $s0, $t0, 4
```

There is, however, one special case in which it is safe, relatively easy, and relatively worthwhile to exploit algebraic properties. Namely, when the left operand is a small constant, the right operand is not a constant, and the

operator is `+`, `*`, or any of the relational operators, it pays to turn the operation around to make use of the immediate mode. (Of course, if the operator is a comparison other than `==` or `!=`, you'll also have to reverse the comparison when you switch the operands.) The word "small" above (in "small constant") refers to the fact that the MIPS hardware only supports constant that fit within 16 bits, so there is no performance to be gained by re-ordering the operands to put the constant on the right if it is bigger than 16 bits. You could go ahead and do it anyway, though—there is no harm.

Make your compiler do these sorts of optimization. You may want to first do everything except targeting (so that you have one more instruction for each of the above example results), and then once that is working add targeting.