MC97 Lab 1: Your first compiler

Due February 19, 1996; Max Hailperin

Your first compiler will compile a single simple arithmetic expression into an assembly language program for the MIPS processor which computes the value of that expression and prints it out.

The expressions your first compiler should be able to handle are all those built up (in infix format) from operands that are single digits using the operators `+` and `-` and using parentheses. For example, `5-(9+2)-4` would be an expression in this language. Given this expression, your program will generate a MIPS assembly language program which computes the answer (-10) and prints that out. [Note that your compiler shouldn't do the arithmetic at compile time. That is a valuable optimization in real compilers, called *constant folding*. In this simple example, however, it would take all the fun out of the compiler.]

Your compiler should be based on Figure 2.24 in the textbook (page 55). The language we are compiling is slightly more complicated, in that we are allowing parenthesized subexpressions. Therefore, a good first step would be to add this language feature to the original translation scheme of Figure 2.19, page 48. Does this have any non-trivial impact on the result of left-recursion elimination, translation scheme 2.14 on page 50? Now, to be sure you are in control, make the corresponding change to Figure 2.24, and try out the resulting program. At this point, you should still be generating postfix expressions as your output, rather than MIPS assembly language, but at least you are compiling the right language.

The next step is to modify your compiler to generate MIPS output. The crucial question is what the appropriate assembly language output to produce is for the computation and for printing out the result. This is most easily shown by example. Here is the code for `5-(9+2)-4`; while reading this keep in mind the postfix form, `5 9 2 + - 4 -`:

```
main:
        li      $t0, 5
        li      $t1, 9
        li      $t2, 2
        add     $t1, $t1, $t2
        sub     $t0, $t0, $t1
        li      $t1, 4
        sub     $t0, $t0, $t1
        move    $a0, $t0
        li      $v0, 1
        syscall
        li      $v0, 0
        j       $ra
```

To move a constant into a register, we use the `li` (add immediate) instruction. Thus, `li $t0, 5`, moves `5` into the zeroth temporary register, which is written `$t0`.

To add or subtract two registers, we use the `add` or `sub` opcode. For example, `sub $t0, $t0, $t1` means `$t0 := $t0 - $t1`.

To print out a number, we need to do three things:

1. We move the value to print into register `$a0`, which is the zeroth argument register, using `move`.

2. Next we put into register `$v0` the constant value 1 as an indication that what we want done with the value in `$a0` is for it to be printed as an integer, i.e., we want to do the "print integer" system call. This is achieved with an `li` instruction.

3. Finally, we do a `syscall` instruction to actually make the system call happen.

After having printed out the result number, the only thing left for your program to do is to halt by returning from the `main` procedure to the simulated operating system with the program's return value being 0, indicating normal completion. The final two lines shown above, starting with `li $v0, 0`, arrange for this normal termination. The `li` line is what makes the return value 0; it works because register `$v0` is used by convention for return values from procedures (including the `main` procedure). The final line, with the `j` instruction is the actual returning from the procedure.

The one other thing you will need to know about the MIPS architecture is what registers you have to work with. MIPS has 32 integer registers, called `$0` through `$31`. However, there are also symbolic names for them that reflect their conventional uses. The ten that are conventionally used for temporary storage are known as `$t0` through `$t9`. Therefore, these are the ones you should use.

Note that to simulate the stack-based evaluation of a postfix expression, we are using the registers in a stack-like fashion. First we "push" 5 into `$t0`, then 9 into `$t1` and 2 into `$t2`. Next we do the addition by "popping" the top two stack elements (`$t2` and `$t1`), adding them, and "pushing" the result back on top of the stack at `$t1`. We aren't really using a stack, we are using general purpose registers which can be accessed in any arbitrary pattern. However, we are (for now) choosing to use them as though they were a stack.

When you have gotten your compiler to produce assembly code that looks reasonable by eye, it is time to try actually running it. Put the compiler's output into a file, e.g. `foo.s`. (One way you could do this is by using the shell's output redirection operator `>`, as in `myCompiler >foo.s`.) Then use the load command within the `spim` program to load in your assembly file (e.g., `foo.s`), and the run command to run it.

You should turn in your program; if the assembly output it generates has a form different from what is shown above, turn in a sample of the output as well.

Also, think about what improvements would be possible in the generated code, even without doing the arithmetic at compile time. The example shown above was designed more for simplicity of generation than for efficiency of execution. Rewrite it by hand as concisely as possible without doing any of the arithmetic. (The SPIM documentation includes a description of the MIPS instruction set, so you know what is possible.) All you need to hand in is your hand rewriting—you don't need to actually modify your compiler to generate the more efficient code. However, it would be good to start thinking about how you would do that.