MC97 Lab 2: Adding Lexical Analysis

Due March 1, 1996; Max Hailperin

You only have to turn in the final compiler you have at the end of this lab. However, I strongly urge you to stop at the end of each paragraph (except this one) and test to make sure you have a working compiler.

Use `lex` to create a lexical analyzer routine that skips white space and comments from `//` to the end of the line and which recognizes unsigned integers and the symbols `+`, `-`, `(`, and `)`. It should return an indication of which of these five kinds of tokens it has read and in the case of integers store the value into a global variable. Integrate this with your compiler from the first lab, so that you now have a somewhat more advanced compiler that can handle multi-digit numbers and ignore comments and whitespace. A sample makefile that may help you manage the components of this compiler is in `~max/MC97/Makefile`. That sample makefile assumes you have a symbol table component, which you don't yet, so you'll need to modify the makefile a bit. (The sample Makefile is also attached to this handout.)

Next extend your lexical analyzer to recognize the keyword `print` and the symbol `;`. Change the parser to expect a program consisting of a list of statements, where each statement consists of `print`, an expression, and then a semicolon. (Be sure that you pop the printed value off of the conceptual stack of registers, so that the register is available for re-use in the next statement. Otherwise a long list of simple print statements would exhaust all the available registers.) Remember that in the sample assembly language output from lab 1, everything up through the `nop` is part of the printing, while the last five lines are the return from `main`, and hence should appear only once at the end of your output.

Now extend the lexical analyzer to recognize `=` as an assignment operator and to recognize identifiers. (In our language, we will allow identifiers to start with a letter or underscore and then have zero or more additional characters which can be digits as well as letters or underscores.) Add symbol-table handling for identifiers, building on the templates in `~max/MC97/symtab.h` and `~max/MC97/symtab.c`, copies of which are attached. The structures stored in the symbol table should have room, in addition to the name, for a flag indicating whether the variable has been assigned to yet and for a register number. Change the parser to allow identifiers in expressions and to allow a second kind of statement, namely an assignment statement. The first assignment statement for a particular variable should allocate a register from the range `$s0` through `$s7` and record this in the structure. All assignments, whether the first or otherwise, should pop the value off the stack and move it into the allocated register. Any time the variable appears in an expression, you should report an error if it hasn't been assigned to; otherwise, copy the value from the variable's register to the stack top.

Once your compiler can compile an arbitrary sequence of assignment statements and print statments, add multiplication and division, using the `mul` and `div` opcodes in the assembly output.