SimpleCalc
CSCI-400 Spring 2013

Preliminary Work

Run Flex and Bison on the SimpleCalc.l and SimpleCalc.y files, respectively, and compile the resulting C files into an executable. Run the executable (it takes no command line arguments) and play with it. Make sure you understand how its behavior maps to the contents of the Flex and Bison input files.

Exercise #1

Modify SimpleCalc so that it accepts an optional command line argument specifying a text file containing the input. If no argument is supplied, the input should be taken from the command line.

Modify the files so that the proper precedence is observed and so that it supports addition, subtraction, multiplication, and division. Refer to Chapter 4 of the text if you need to brush up on how the grammar can enforce precedence.

The program should output something each time a token is recognized and each time a production rule is reduced. The output for a token should be along the lines of:

TOKEN: NUMBER (4.59 from “4.59”)

The first 4.59 is the lexeme as passed to the parser, the second 4.59, in quotes, is the string of characters from the source that produces the lexeme. They will often, but not always, be the same.

The output for a grammar reduction should be along the lines of:

REDUCE: <expression> -> <expression> + <term> (4.59)

The specific production rule used should be discernible as well as the value associated with the new non-terminal.

Exercise #2

Modify the files so that SimpleCalc can work with either integers or floating point values. Refer to the slides presented in class for details on switching from using yylval defined as an external global integer to using a union defined in the Bison file. Remember that you now have to declare which element in the union is used for each token type and each non-terminal type. Your grammar should keep integers as integers as long as possible. So, for instance, if two integers are added together or divided, the result should be an integer. But the result of combining an integer and a floating point value should be a floating point value.

SimpleCalc
CSCI-400 Spring 2013

Exercise #3

Modify SimpleCalc so that it accepts assignment statements of the form

```
R4 = 3.4*5
```

To get some exposure with preprocessing lexemes, accept identifiers in which the first character is an R (uppercase or lowercase) followed by any number of letters or underscores and ending with a single digit. The lexeme that is passed to the parser should consist of just the leading capital R and the trailing digit. Thus,

```
register_4 = 3.4*5
```

would be effectively identical to the earlier statement.

Exercise #4

Modify SimpleCalc so that it accepts a program (loosely define) consisting of a sequence of expressions enclosed by curly braces. For instance

```
{
   register_4 = 3.4*5;
   R9 = 5 + 93/7;
   reg6 = 7 + 4.32;
}
```

You will need to add a non-terminal that recognizes an entire program. But it should also still be able to process the single statements used by the previous exercises.

Note that you do not (yet) have to do anything with values assigned to registers, which means that you can't (yet) use these values in the expressions. In fact, do not need to store any of the results at all. At this point you are still primary concerned that things are being recognized and processed correctly. However, it might be useful to know that, in the next step, registers R0 through R4 will be dedicated as integers and R5 through R9 will be dedicated as floating point.

Exercise #5

Modify SimpleCalc so that the values stored in registers can be used in subsequent expressions. This would normally involve using a symbol table, but we can simplify this significantly since we know that we have at most ten variables and they are named R0 through R9. As noted earlier, R0 through R4 are integer variables and R5 through R9 are floating point variables.

To keep things simple, we can deal with the issue of uninitialized variables by simply initializing all variables to a value of zero.

If the input was a single statement, your program should print out the value of the expression or the value assigned to the variable, as appropriate. If the input was a program, your program should finish by printing out a table of all of the values stored in all of the registers.

Exercise #6

Modify SimpleCalc so that it supports exponentiation via the '^' operator. The result of any exponentiation should be a floating point value, even if both the base and the exponent are integers and even if the result can be exactly represented by an integer variable.

Submission

Submit your zipped Flex and Bison input files to BlackBoard. Only submit a single set of input files, named SimpleCalc.l and SimpleCalc.y (capitalization doesn't matter); do NOT submit one for each exercise. The grading will be done by running a batch file to process your input files, compile the result, and run against a test input file. Be sure to test your program using very small and also very large floating point values.

GRADING RUBRIC – 80 pts

20	**- Good Faith effort.**
10	**- Exercise #1**
10	**- Exercise #2**
10	**- Exercise #3**
10	**- Exercise #4**
10	**- Exercise #5**
10	**- Exercise #6**
-10	**- Improper submission.**