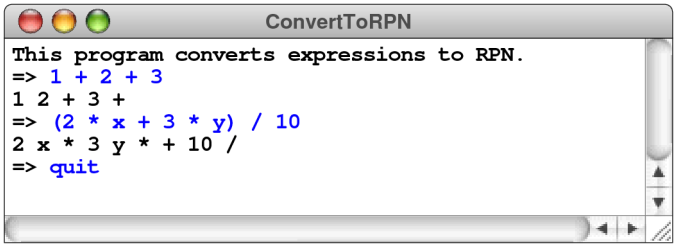
Eric Roberts
CS106B

Handout #48
March 4, 2013

Section Handout #8
Expressions

1. Convert an expression to Reverse Polish Notation

Write a program that reads expressions from the user in their standard mathematical form and then writes out those same expressions using Reverse Polish Notation, in which the operators follow the operands to which they apply. (Reverse Polish Notation, or RPN, was introduced in the discussion of the calculator in Chapter 5.) Your program should be able to duplicate this sample run:



Your program should work entirely as a client of the **`exp.h`** interface.

2. Constant folding (Chapter 19, exercise 13, page 881)

After it parses an expression, a commercial compiler typically looks for ways to simplify that expression so that it can be computed more efficiently. This process is called ***optimization***. One common technique used in the optimization process is ***constant folding***, which consists of identifying subexpressions that are composed entirely of constants and replacing them with their value. For example, if a compiler encountered the expression

```
sec = 24 * 60 * 60 * days
```

there would be no point in generating code to perform the first two multiplications when the program was executed. The value of the subexpression **`24 * 60 * 60`** is constant and might as well be replaced by its value (86400) before the compiler actually starts to generate code.

Write a function **`foldConstants(exp)`** that takes a pointer to an **`Expression`** and returns a pointer to an entirely new expression in which any subexpressions composed entirely of constants are replaced by the computed value.

3. Changing the interpreter into a compiler (Chapter 19, exercise 14, page 881)

Although the interpreter program that appears in this chapter is considerably easier to implement than a complete compiler, it is possible to get a sense of how a compiler works by defining one for a simplified computer system called a ***stack machine***. A stack machine performs operations on an internal stack, which is maintained by the hardware,

Figure 1. Stack machine instructions

LOAD #*n*	Pushes the constant *n* on the stack.
LOAD *var*	Pushes the value of the variable *var* on the stack.
STORE *var*	Stores the top stack value in *var* without actually popping it.
DISPLAY	Pops the stack and displays the result.
ADD **SUB** **MUL** **DIV**	These instructions pop the top two values from the stack and apply the indicated operation, pushing the final result back on the stack. The top value is the right operand, the next one down is the left.

in much the same fashion as the calculator described in Chapter 5. For the purposes of this problem, you should assume that the stack machine executes the operations shown in Figure 1.

Write a function

```
void compile(istream & infile, ostream & outfile);
```

that reads expressions from **infile** and writes to **outfile** a sequence of instructions for the stack-machine that have the same effect as evaluating each of the expressions in the input file and displaying their result. For example, if the file opened as **infile** contains

```
x = 7
y = 5
2 * x + 3 * y
```

calling **compile(infile, outfile)** should write the following code to **outfile**:

```
LOAD #7
STORE x
DISPLAY
LOAD #5
STORE y
DISPLAY
LOAD #2
LOAD x
MUL
LOAD #3
LOAD y
MUL
ADD
DISPLAY
```