

CS2604 (Fall 2001)

PROGRAMMING ASSIGNMENT #1

Due Wednesday, September 19 @ 11:00 PM for 100 points

Early bonus date: Tuesday, September 18 @ 11:00 PM for 10 point bonus

Late penalty: Thursday, September 20 at 11:00 PM for 30 point penalty

Assignment:

In most programming languages, integer values are of a fixed size, e.g., 32 bits in **C++**, 64 bits in Java. 32 bits will allow integers of approximately 10 decimal digits. For most purposes this is sufficient, but when it is not, infinite precision integer arithmetic can be implemented in software. For this project you will implement a “bignum” or infinite precision arithmetic package for non-negative integers.

Input and Output:

The input to this program should be **read from standard input** and the output should be **directed to standard output**. The name of the program should be “bignum”.

The input file will consist of a series of arithmetic expressions. To simplify the project, all expressions will be in Reverse Polish Notation (RPN). In RPN, the operands come before the operator. For example, you normally see multiplication written $(a * b)$. In RPN, this expression would appear as $(a b *)$. For expressions with more than one operator, each time an operator is encountered, it operates on the two immediately preceding sub-expressions. For example, $(a + b * c)$ in “normal” notation would be $(a b c * +)$ in RPN. This means that b and c are multiplied, then the results of the multiplication are added to a (this follows the normal rules of arithmetic precedence in which $*$ operations take precedence over $+$ operations). In RPN, there is no operator precedence; operators are evaluated left to right. Thus, the expression $((a + b) * c)$ is written $(a b + c *)$ in RPN.

Input operands consist of a string of digits, and may be of any length. You should trim off any excess zeros at the beginning of an operand. Expressions may also be of any length. Spaces and line breaks may appear within the expression arbitrarily, with the rule that a space or line break terminates an operand, and a blank line (i.e., two line breaks in a row) terminates an expression. The operations to be supported are addition ($+$), multiplication ($*$), and exponentiation ($^$). An error should be reported whenever the number of operators and operands don't match correctly – i.e., if there are no operands left for an operator, or if the expression ends without providing sufficient operators. When an error is detected, processing should stop on the current expression, and your program should proceed to the next expression.

For each expression, echo the input as it is read, followed by the result of the operation, or an error message, as appropriate. Be sure that the result is clearly marked on the output for readability.

Implementation:

RPN simplifies the project since it can be easily implemented using a stack. As the expression is read in, operands are placed on the stack. Whenever an operator is read, the top two operands are removed from the stack, the operation is performed, and the result is placed back onto the stack. You may assume that the stack will never hold more than 100 operands.

The main problem in this project is implementation of infinite precision integers and calculation of the arithmetic operations $+$, $*$, and $^$. **NOTE:** These three operations must be implemented **recursively**. Integers are to be represented in your program by a linked list of digits, one digit

per list node. You must maintain a freelist of linked list nodes as discussed in class, and allocate new link nodes with the **new** operator as needed. Each digit may be represented as a character, or as an integer, as you prefer. You will likely find that operations are easier to perform if the list of digits is stored backwards (low order to high order).

Addition is easily performed by starting low order digits of the two integers, and add each pair of digits as you move to the high order digits. Don't forget to carry when the sum of two digits is 10 or greater!

Multiplication is performed just as you probably learned in elementary school when you multiplied two multi-digit numbers. The low order digit of the second operand is multiplied against the entire first operand. Then the next digit of the second operand is multiplied against the first number, with a "0" placed at the end. This partial result is added to the partial result obtained from multiplying the previous digit. The process of multiplying by the next digit and adding to the previous partial result is repeated until the full multiplication has been completed.

Exponentiation is performed by multiplying the first operand to itself the appropriate number of times, decrementing the exponent once each time until done. In order to simplify implementation, you are guaranteed the the exponent will be small enough to fit in a regular `int` variable. You should write a routine to convert a number from the list representation to an integer variable, and use this to convert the top operand on the stack when the exponentiation operator is encountered. Be careful with exponents of 0 or 1.

You are permitted to use in your program any code from the textbook (e.g., list and stack classes). **Warning:** the textbook code was not written for this purpose, and is likely to need modification. But you may use it to get you started if you think it will help.

You may NOT use in your program the list or stack classes provided with your compiler system, or from any other source.

Programming Standards:

You must conform to good programming/documentation standards, as described in the Elements of Programming Style. Some specifics:

- You must include a header comment, preceding `main()`, specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.

- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation, unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point. You may not use code from STL, MFC, or a similar library in your program.

Testing:

A sample data file will be posted to the website to help you test your program. This is not the data file that will be used in grading your program. While the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

Deliverables:

You will submit this project electronically. In particular, you will create a zip'ed archive file containing the following items (and nothing else):

- all source code files necessary to build an executable
- either the project workspace files (.dsw and .dsp) for Visual C++ users, or a makefile for g++ users.

Windows users should be sure to use a modern zip tool which preserves long file names. A suitable freeware command-line zip tool will be posted on the course website. UNIX users should submit a gzip'ed and tar'ed file.

Once you have assembled the archive file for submission, for your own protection, please move it to a location other than your development directory, unzip the contents, build an executable, and test that executable on at least one input file. Failure to do this may result in delayed evaluation of your program, and a loss of points.

You will submit your project to the automated Curator server. The instructions and necessary software are available at: <http://ei.cs.vt.edu/~eags/CuratorGuides.html>. If you make multiple submissions, only your last submission will be evaluated.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement in the header comment preceding the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used C++ language code obtained from another student,
//   or any other unauthorized source, either modified or unmodified.
//
// - All C++ language code and documentation used in my program
```

```
// is either my original work, or was derived, by me, from the source
// code published in the textbook for this course.
//
// - I have not discussed coding details about this project with anyone
// other than my instructor, ACM/UPE tutors or the GTAs assigned to this
// course. I understand that I may discuss the concepts of this program
// with other students, and that another student may help me debug my
// program so long as neither of us writes anything during the discussion
// or modifies any computer file during the discussion. I have violated
// neither the spirit nor letter of this restriction.
//
```

Programs that do not contain this pledge will not be graded.