CSCI 6010 Introduction to Computer Science Fundamentals, Fall 2016

Project 1: Due on midnight, Oct. 6, 2016

Infinite Precision Arithmetic

Assignment: In most programming languages, integer values are of a fixed size. 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* (This is a type of data structure with the key feature of Last-In-First-Out, called LIFO; you can use *Stack* class in Java for common methods such as *pop*, *push*, *search*, etc.). 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 can be implemented *recursively*,

meaning that one operation can be called in another operation (e.g., * can be performed with + and ^ can be performed with *). Integers are to be represented in your program by an array of digits, one digit per array slot. You may want to maintain a new array list 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 can be 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 that the exponent will be small enough to fit in a regular *int* variable. You may want to write a subroutine (method) 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.

Testing examples:

(1)

Input: 000000056669777 99999911111 + 352324012 + 3 ^ 555557778 *

Output: (((56669777+99999911111)+352324012) ^ 3) * 555557778 = 562400792227677956625810678708149922000000.

The output should omit the first number's 0, and omit the extra space and line break.

(2)

Input: 99999999 990001 * 01119111 55565 33333 + * + 88888888 +

Output: ((99999999*990001) + (01119111 * (55565+33333))) + 88888888 =

99099674628565

(3)

Input: 123456789 1111111111 * 111119 2111111 9111111 * + * 1 ^

(This is to test if the program can handle 1 as an exponent.)

Output: ((123456789*1111111111) * (111119+(2111111*9111111)))^1=

263848650047763865232585126976 0

(4)

Input: 9 1 + 5 * 0 +

Output: (9+1) * 5 + 0 = 50

(5)

Input: 999999999 0 *

(This tests if the program can handle multiplying by zero.)

Output: 999999999 * 0 = 0.

(6)

Input: 9 0 ^

(This tests if the program can handle a zero exponent.)

Output: 9^0 = 1

(7)

Input: 5555555 333333 5454353 999999 666666 01 ^ * * +

(This tests the error of insufficient operators.)

Output: Error

The numbers remaining in the stack are 3636228060866636235 and

5555555.

(8)

Input: 3432 3333 9999 + * ^ * * 6666 +

Output: Error (This tests the error of insufficient operands. The program should ignore the remainder of the input stream until a new expression begins.)

Deliverables: One zip file named 'P1_(your_last_name)_(your_first_name).zip' (e.g., P1_Doe_John.zip) including all your source and executable files, which are *.java* and *.class* files. All input values are read from an input file and the output should be printed out on the console. Note that you should have three files under the directory named 'P1_(your_last_name)_(your_first_name)' as follows:

bignum.java

bignum.class

input.txt

where input.txt is an input file which is read from your program and used to test correctness of your program. An example input.txt is provided in the blackboard with a file name 'p1.txt'.

Your program will be tested in C:\....\ P1_(your_last_name)_(your_first_name) by

java bignum

Any failure to test your program in this setting will result in point deductions. In particular, if your program needs to be fixed for proper compilation and run, this will result in 5 point deductions. Note that an instructor is not responsible for debugging your program.

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 in your main file (bignum.java) as follows:

"I pledge that this assignment has been completed in compliance with the Graduate Honor Code and that I have neither given nor received any unauthorized aid on this work. Further, I did not use any source codes from any other unauthorized sources, either modified or unmodified. The submitted programming assignment is solely done by me and is my original work."

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

Reminder:

Assignment Policy: a late assignment turned in will be penalized 10% off for each day late. After a week late, the late homework will be 100% off. For programming assignments, Java is a preferred programming language to use. If you want to use other programming language, please discuss with the instructor before starting your programming assignment.

If you need an extension of the due because of any emergency situation, you should discuss with the instructor in prior to the project due date.