CS217, Spring 1998 Assignment 7 BigNum Calculator

Due : April 26, 11:59 pm

Implement an arbitrary precision reverse polish notation calculator on big numbers. Reverse polish notation is a notation for describing arithmetic expressions in which the operations appear after the arguments. For example to denote the expression `x + y` one would write: `x y +`, and for the expression `(x+y)*z` is denoted by `x y + z *`, where `x, y, z` stand for numbers.

The calculator can only be used in an interactive fashion i.e. it always reads from stdin. The format of its inpout is one token per line only. For the second expression above the input would look like:

```
x
y
+
z
*
```

Reverse polish notation is popular among compiler writers because an evaluator (calculator) for its expressions can easily be implemented with the help of a stack.

The calculator needs to support the following operations:

- $+$: The addition operator. It pops the two top BigNums from the evaluation stack, performs the operation, pushes the result on the stack and frees the two operands.
- $-$: The subtraction operator. It pops the two top BigNums from the evaluation stack, performs the operation, pushes the result on the stack and frees the two operands.
- $*$: The multiplication operator. It pops the two top BigNums from the evaluation stack, performs the operation, pushes the result on the stack and frees the two operands.
- $/$: The division operator. It pops the two top BigNums from the evaluation stack, performs the operation, pushes the result on the stack and frees the two operands.
- $\%$: The remainder operator. It pops the two top BigNums from the evaluation stack, performs the operation, pushes the result on the stack and frees the two operands.
- e : The exponentiation operator. It pops the two top BigNums from the evaluation stack, performs the operation, pushes the result on the stack and frees the two operands.

- $=$: The equality operator. It pops the two top BigNums from the evaluation stack, compares them, pushes the BigNum 1 if they are equal or the BigNum 0 if they are different on the stack and frees the two operands.
- $>$: The greater than operator. It pops the two top BigNums from the evaluation stack, compares them, pushes the BigNum 1 if the first is greater than the second or BigNum 0 otherwise on the stack and frees the two operands.
- $<$: The greater than operator. It pops the two top BigNums from the evaluation stack, compares them, pushes the BigNum 1 if the first is less than the second or BigNum 0 otherwise on the stack and frees the two operands.
- n : The negation operator. It pops the top element off the stack, negates it, pushes the result on the stack and frees the operand.
- d : The duplicate operator. It duplicates the BigNum on top off the stack. For example to add 1 + 1 and get two I may do:

  ```
  1
  d
  +
  ```

  The stack should now only contain the BigNum 2.
- p : The print_top operator. It prints to stdout the top element on the stack followed by a newline.
- f : The print_top operator. It prints to stdout every elements on the stack starting from the bottom of the stack. Each element is followed by a newline. If the stack is empty the operator prints absolutely nothing.

With the exception of the f operator, if the stack does not contain enough arguments then the calculator should print out the following string to stdout : `"Not enough arguments.\n"`. Following that it should `exit(1)`. The calculator exits normally if the `EOF` is reached i.e. the user types CTRL-D.

Big numbers

Now the only problem is that an int on our computer can hold only a 32-bit (10 decimal digit) number, not a 200-digit number. So how do we do exponentiation and modulus on these big numbers? Let's represent a number as an array of digits in the base b. You are used to base 10, of course; but we will use base $b = 65536$ since that's convenient on the computer. An unsigned short can hold numbers in the range 0..65535, so a 200-digit decimal number is just an array of about 80 unsigned shorts. Or, with base 4294967296, an unsigned long can hold each digit. These numbers can be added, subtracted, and multiplied by the same rules you learned in grade school (before they let you use pocket calculators). Remember carries and borrows? Just start at the low-order digit and work your way up. To do the mod operation, we need to do long division and keep the remainder. Here are some mathematical rules regarding division:

$$a < b \rightarrow a/b = 0 \ \ rem \ \ a$$
$$a/2b = q \ \ rem \ \ r, r < b \rightarrow a/b = 2q \ \ rem \ \ r$$
$$a/2b = q \ \ rem \ \ r, r \geq b \rightarrow a/b = 2q + 1 \ \ rem \ \ r - b$$

We also have the problem of exponentiating numbers. To raise a to the power k, all we have to do is multiply a by itself k times. The problem is that if $k = n - 1$, and n is a 200-digit number, this will take much longer than the age of the universe. But we can use the following identities:

$$k \ \ even \rightarrow a^k = (a^{k/2})^2$$
$$k \ \ odd \rightarrow a^k = a \cdot a^{k-1}$$

Your assignment

In the precept section, discuss how to modularize this program and define the interfaces between modules. Discuss the mathematical algorithms for each of the You must rigorously test your modules, since bugs in your modules will cause the whole program to fail. Methods for testing will be discussed in precept. Make sure to put in assertions. Part of what you turn in should be a description of how you tested, or otherwise ensured the correctness, of your modules. It is not cheating to test together with someone who is working on a different module. You must ensure, of course, that you have been responsible for designing and implementing your own module. But testing might be much easier if you can hook more than one module together.

Grading

Your grade for this program will be based on your participation in precept, on the test strategy you have designed, on the correctness of your program, and on the elegance and readability of your program and documentation.

Submission

Submit your two modules and a description of how you tested them using the command:

```
/u/cs217/bin/submit 7 ....
```

Notes

It couldn't hurt for you to understand the algorithms used by the modules you don't have to implement. You will have to do something about freeing the bignums that are results of intermediate calculations, or else they will eat up all of memory.