CS 6301.502. Implementation of advanced data structures and algorithms

Fall 2017; Wed, Sep 6.

Long Project LP1: Integer arithmetic with arbitrarily large numbers

Ver 1.0: Initial description (Wed, Sep 6).

Ver 1.1: Changed type of base to long in Num.java

Ver 1.2: Added missing line in grammar (level 4) for operator % (Tue, Sep 12)

Due: 11:59 PM, Sun, Sep 24 (1st deadline), Sun, Oct 8 (2nd deadline).

Max excellence credits: 10.

* Submit before the first deadline (Sun, Sep 24) to be eligible for excellence credit.
* For each group, only its last submission is kept and earlier submissions are discarded.
* Your code must be of good quality, and pass all test cases to earn excellence credits.
* Only the first 10 groups satisfying above conditions will be assigned excellence credits.

**Project Description**

In this project, develop a program that implements arithmetic with large integers, of arbitrary size.

Code base: Java library: Lists, stacks, queues, sets, maps, hashing, trees. Do not use BigInteger, BigNum, or other libraries that implement arbitrary precision arithmetic.

[Starter code and sample drivers are provided. Download lp1-starter.zip here](https://www.utdallas.edu/~rbk/teach/2017f/projects/lp1-starter.zip)

Your task is to implement the class Num that stores and performs arithmetic operations on arbitrarily large integers. You must use the following data structure for representing Num: Linked list or Array List or Array of long integers, where the digits are in the chosen base. In particular, do not use strings to represent the numbers. Each node of the list stores exactly one long integer. The base is defined to be 10 in the starter code, but you may modify it. In the discussions below, we will use base = 10, using linked lists to represent Num. For base = 10, the number 4628 is represented by the list: 8-->2-->6-->4.

**Level 1**

Implement the following methods:

* Num(String s): Constructor for Num class; takes a string s as parameter, with a number in decimal, and creates the Num object representing that number in the chosen base. **Note that, the string s is in base 10, even if the chosen base is not 10.** The string s can have arbitrary length.
* Num(long x): Constructor for Num class.
* String toString(): convert the Num class object into its equivalent string (in decimal). There should be no leading zeroes in the string.
* Num add(Num a, Num b): sum of two numbers stored as Num.
* Num subtract(Num a, Num b): given two Num a and b as parameters, representing the numbers n1 and n2 repectively, returns the Num corresponding to n1-n2.
* Num product(Num a, Num b): product of two numbers.
* Num power(Num x, long n): given an Num x, and n, returns the Num corresponding to x^n (x to the power n). Assume that n is a nonnegative number. Use divide-and-conquer to implement power using O(log n) calls to product and add.
* printList(): Print the base + ":" + elements of the list, separated by spaces.

**Level 2**

* Num power(Num x, Num n): return x^n, where x and n are both Num. Here x may be negative, but assume that n is non-negative.
* Num divide(Num a, Num b): Divide a by b result. Fractional part is discarded (take just the quotient). If b is 0, raise an exception.
* Num mod(Num a, Num b): remainder you get when a is divided by b (a%b). Assume that a is non-negative, and b > 0.
* Num squareRoot(Num a): return the square root of a (truncated). Use binary search. Assume that a is non-negative.

You can get 100 points for the project by completing levels 1 and 2. Levels 3 and 4 are for scoring excellence credits.

**Level 3**

Write a driver program that illustrates the methods, based on the following input/output specification. The operands of the expressions are names of variables (single lower case letter, a-z) and integer constants (0 or [1-9][0-9]\*). Note that there is no unary minus operator. The operators are {+, -, \*, /, %, ^, |} representing add, subtract, multiply, divide, mod, power, and square root, respectively. Tokens will be separated by spaces, and each expression will be ended by a semicolon. Each input line will have the form:

**var ;**

**var = postfix expression ;**

**;**

When processing the first line above, your program should just output the value of the variable. Other lines will have a postfix expression being assigned to a variable. Your program should evaluate the expression, store it, and print its value to the console. If a line with just a semicolon is encountered, then your program should stop processing, and ignore any additional lines in the input. At the end, output the internal representation of the last variable that was assigned a value. Here is a sample input and its output:

Sample input:

a = 999 ;

b = 8 ;

c = a b ^ ;

d = a b + ;

;

Its output:

999

8

992027944069944027992001

1007

10: 7 0 0 1

**Level 4**

Complete levels 1-3 and the following additional functionality. Let the base have a default value, but allow it to be changed by passing a value in the command line. Your programs should continue to work, for any choice of base, between 2 and 10,000. For example, if your program is run as "java LP1L4 32", then all numbers should be stored in base 32. **Note that input/output is always in base 10.**

Implement the [Shunting Yard algorithm: https://en.wikipedia.org/wiki/Shunting-yard\_algorithm](https://en.wikipedia.org/wiki/Shunting-yard_algorithm) to be able to parse arithmetic expressions using the following precedence rules (highest to the lowest).

* Parenthesized expressions (...)
* Unary operators: square root (|).
* Exponentiation (^), right associative.
* Product (\*), division (/), and, mod (%). These operators are left associative.
* Sum (+), and difference (-). These operators are left associative.

Write a driver program that reads its input from stdin (console). The input is a sequence of lines. Lines that start with the name of a variable are treated the same as level 3, indicating a postfix expression. In addition, some lines will start with a number as its first token. This number is a label that will be used in "goto" statements that control execution. Format for inputs to Level 4 adds the following to Level 3 inputs:

**lineno var = expression ;** # infix expression

**lineno var ? NZ ;** # continue execution from lineno NZ if var != 0

**lineno var ? NZ : ZR ;** # continue execution from lineno NZ if var != 0,

# and from line ZR if var == 0

where expression is a valid arithmetic expression formed by identifiers (variables) and numbers. Nothing should be output when Level 4 input lines are processed. The following grammar generates valid expressions. Note that there is no unary minus operator. Tokens are separated by spaces, so that input can be parsed easily. See [Tokenizer.java](https://www.utdallas.edu/~rbk/teach/2017f/java/Tokenizer.java) for an example.

expression:: expression + term

| expression - term

| term

term:: term \* factor

| term / factor

| term % factor

| factor

factor:: uterm ^ factor

| uterm

uterm:: uterm |

| atom

atom:: var

| number

| ( expression )

var:: [a-z]

number:: 0

| [1-9][0-9]\*

Execution starts from the first line, and proceeds line by line. Lines of the form "lineno var ? NZ ;" and "lineno var ? NZ : ZR ;" change the execution sequence, depending on whether the var is equal to 0 or not. If x = 5, then upon executing the line "10 x ? 8", your program should continue from line with lineno=8. On the other hand, if x = 0, then the line "10 x ? 8" has no effect, and execution continues sequentially. Upon executing the last line, the program comes to a stop.

Assume that the input does not have multiple lines with the same lineno. Line numbers do not have to be sequential or in ascending order. Inputs will not try to access the value of a var that has not yet been assigned a value.

Sample input for level 4, with explanations:

x = 10 ; # Level 3: postfix expression that assigns 10 to x

p = 1 ; # Level 3: p is assigned 1

8 p = p \* x ; # Level 4: expression in infix notation, lineno=8

10 x = x - 1 ; # Level 4: x is decremented by 1

15 x ? 8 ; # Level 4: continue from line with lineno=8 if x != 0

p ; # Level 3: print the value of p

;

The above input calculates 10! by looping between lines with lineno=8 and lineno=15, until x becomes zero. Its output is:

10 # Output from "x = 10 ;"

1 # Output from "x = 10 ;"

3628800 # Output from "p ;"

10: 0 0 8 8 2 6 3 # The last line executed was "p ;", so printList() of var p