UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **M. Brudno**
Summer 2000

Project 1*

Due: Monday, July 3, 2000

This first project involves writing a calculator program that can perform *rational arithmetic*—arithmetic on fractions. We want the program to be able to handle fractions with arbitrarily large numerators and denominators. The really hard part of the job is already done for you: Java has a library type `java.math.BigInteger`, which handles integers of arbitrary size (as Scheme does). You still have to figure out how to use it, of course.

1 Commands

The calculator will be one of the familiar stack variety: to perform the computation $(\frac{2}{3} \times \frac{6}{7} + 8)/7$, for example, one would enter the seven commands

```
2/3  6/7  *  8  +  7  /
```

(on one or more lines, separated by whitespace), and the answer would end up on the top of the stack. The possible commands are as follows:

`#` marks a comment. The `#` character and everything to the end of the line is ignored.

N for N any rational number (see §2 below): pushes N on the calculator's stack.

`=` pops the top of the stack and prints it.

`+, -, *, /, %` pop the top two items on the stack, perform the indicated arithmetic operation (with the top item of the stack on the right of the operator), and push the result back on the stack. Remainder (a%b) for any number a and any $b \neq 0$ (integer or otherwise) is defined as $a - b \cdot q$, where q is the integer resulting from truncating (removing the fraction from) a/b.

`_` (underscore) negates the top element of the stack.

`dup` **or** `d` pushes a copy of the top of the stack.

`exch` **or** `x` exchanges the top two items on the stack.

`round(`D`)` **or** `r(`D`)` pops the top item off the stack, rounds it to the nearest multiple of $1/D$, and pushes the result back on the stack. D must be a positive integer. In case of ties, use the round-to-even rule. That is, if the number to be rounded equals $(N + \frac{1}{2})/D$, the result is N/D if N is even and otherwise $(N + 1)/D$.

*This project was written by Professor Paul Hilfinger

`round` or `r` is the same as **`round(1)`**.

`set proper` Causes numbers to be printed as proper fractions, that is, with an integer part and a fraction with magnitude less than 1. This is the default setting.

`set improper` Causes numbers to be printed as improper fractions—without an integer part.

`trunc(`*D***`)` or `t(`***D***`)`** pops the top item off the stack, truncates it to be a multiple of $1/D$, and pushes the result back on the stack. D must be a positive integer. That is, if the number on the stack is equal to $\pm(N/D + \epsilon)$, where $N \geq 0$ and $0 \leq \epsilon < 1/D$, then the result is $\pm N/D$. For both *trunc* nd *round* remember that D can be of any size.

`trunc` or `t` is the same as **`trunc(1)`**.

`quit`, `q` quit the program.

`?`, `help` prints a help message summarizing these commands.

Commands must be separated from each other by whitespace—that is, by blanks, tabs, or ends of lines (newline characters). The word '`set`' must be followed by blanks or tabs, not newlines. There is no distinction made between upper- and lower-case letters; for example, the command '`t`' is the same as '`T`' and '`quit`' is the same as '`qUiT`'.

2 Format of Numbers

A rational number has one of the following forms (here, $I \geq 0$, $N \geq 0$, and $D > 0$ represent integer numerals):

I A plain integer.

N/D A fraction (not necessarily a proper fraction and not necessarily in lowest terms).

I_N/D A mixed fraction, representing $I + \frac{N}{D}$. For input, we do not require that $|N| < D$.

$-F$ The negation of F, where F is any of the previous three forms.

In all these cases, there may be no embedded blanks.

Numbers are always printed in lowest terms (i.e., with the smallest possible denominator) in one of the formats above. When $N = 0$ or $D = 1$, in particular, the number is *always* printed as an integer (without a fraction part). If `set proper` is in effect, as it is initially, all numbers are printed as proper fractions—that is, as I_N/D or (whenever $I = 0$) as N/D, with $|N| < D$ in either case.

3 Your Task

The directory `~cs61b/hw/proj1` contains a few skeleton files that may suggest some structure for this project. Copy them into a fresh directory as a starting point. Use the command

```
cp -pr ~cs61b/hw/proj1 mydir
```

to create a new directory called `mydir` containing copies of all our files (with the right protections).

I have placed the `Rational` class in a separate package called `mymath`, just to get you used to packages. Feel free to add other classes and files, and to make any modifications you see fit. In fact, you may throw them all away and do your own. To submit your result, use the command '`submit proj1`'. Each partnership should turn in exactly one project (from either partner). Make *sure* that both of your names and logins are on everything you turn in. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time, and list them in your makefile; that makes them very convenient to run.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user tries to perform additions when there is nothing on the stack, or to divide by 0, or asks to '`tound`' rather than to '`round`' a result, your program should not simply halt and catch fire, but should give some sort of message and then try to get back to a usable state. Your choice of recovery is less important than being sure that your program *does* recover. I don't care, for example, whether you pop the two operands off the stack on a division by zero or keep them on the stack, just so long as you do something better than abruptly exit with a cryptic stack trace.

Our testing of your projects will be automated. The grading program will be finicky, so be sure that:

- Your makefile must be set up to compile everything on the command `gmake` and to run all your tests on the command `gmake test`. The makefile provided in our skeleton files is already set up to do this. Be sure to keep it up to date.
- Your main function must be in a class called `RatioCalc`. The skeleton is already set up this way.
- The first line of output of your program identifies the program. It may contain anything.
- Before reading the first command and on reading each subsequent end-of-line, your program must print the prompt '`>␣`' (greater than followed by a blank). No cute or obscene phrases, please, just '`>␣`'.
- Output numbers in exactly for format specified in §2 above, with no extra adornment.
- Put your error messages separate lines, starting with the word '`ERROR`'. The grading program will ignore the text of the message.
- Your program should exit without further output when it encounters a `quit` command or an end-of-file in the input.
- Your final version must not print any debugging information.

4 Advice

You will find this project challenging enough without helping to make it harder. Much of what might look complicated to you is actually pretty easy, once you take the time to look to see what has already been written for you—specfically what is in the standard Java library of classes. So, before doing any hard work on anything that looks tricky, browse your texts and the documentation. Naturally, the `java.math.BigInteger` class will be useful. (For example, I wonder what the `gcd`

function might be good for. Despite its documentation, it returns the greatest common *divisor,* not the greatest common denominator, whatever *that* might be.)

For reading input from the user, I expect that you'll find `java.io.StreamTokenizer` and `java.util.StringTokenizer` to be useful as well. The standard Java string type, `java.lang.String`, also contains useful member functions for doing comparisons (including comparisons that ignore case). Read the on-line documentation for all of these. You will *not* be able to use the `parseNumbers()` feature of `StreamTokenizer` [why not?].

Finally, because of the nature of the summer course, you do not have a lot of time to write this project. I suggest you start looking at it immediately, and start coding soon.

5 Sample Session

User input is underlined.

```
% java RatioCalc
RatioCalc, version 1.0
> # Here is the example from earlier
> 2/3 6/7
> * 8 + 7 / =
1_11/49
> q
```