Contents

1 Overview

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Project1, put each program in its own subdirectory, compress this into a single zip file, and then submit this zip file to the Project #1 assignment. Also remember to create make files for each program and make sure the programs compile and run on the lab Linux system.

Projects are to be done strictly on your own and as with all assignments the sharing of files and code is strictly prohibited and constitutes an act of Academic Misconduct. Furthermore the use of any electronic medium, such as code repositories, forums, blogs, message boards, email, etc. is strictly prohibited and constitutes an act of Academic Misconduct.

You may use the course textbooks, class notes, and materials I have provided on the MyClasses page.

This exercise is for you to build a workspace-style system that will preform exact integer arithmetic with unlimited length integers. This is similar to the way that computer algebra systems work, for example Mathematica and Maxima. Although creating an arbitrary length integer in C++ is a good exercise in storage, mathematics, and operator overloading, we will use one that is open-source and freely distributed on-line, InfInt which is included in the support files for this project.

Our goal is to write an expression tree implementation for processing any valid mathematical expressions using our defined operations and integer terminals. The program will take in a string of the expression from the user, parse the string and create an expression tree from it. After the expression tree is created it will recursively evaluate the tree and report the resulting value. In the process, you will also track syntax errors and evaluation errors in the user input. Errors will be displayed in place of numeric results if one exists.

In addition, you will incorporate file saving and loading of the current workspace. Here we will use another third-party open-source package called tinyfiledialogs (included in the support files) that will allow dialog based file selection of filenames and paths for easier loading and saving of your files. The tinyfiledialogs package is a cross-platform package that will easily invoke the system-level standard dialog boxes for file opening and saving, among other things.

Before you begin, make sure that you understand the parsing and evaluation portions of the expression tree example given in class. Since these expressions will be all numeric, you do not need to completely understand the symbolic differentiation portion of the code for this assignment. You will be using the general structure of the expression tree code, but with significant changes.

2 Programs

2.1 Arbitrary Length Integer Calculator

This program will have a workspace style interface that will continually take input from the user in string form. The strings will represent mathematical expressions that incorporate standard integer operations and a few integer based functions. There will be no decimal approximations or variables in the expressions. You will also be revising the evaluate function to handle these types of expressions. Since there are no variables, you will not be implementing a differentiation function, which is the bulk of the code for the expression tree example. Just to give you an idea here, my implementation of this tree class took about 300 lines of code (not the 1200 of the example). Even though you are using the example as a basis for your program I do not want any unnecessary functions, so all the derivative and most of the function code and support functions for them are to be removed.

You will be using two third party packages that are included as support files for this project. One is the infinite precision arithmetic package InfInt.h. This is a single header file implementation of an arbitrary precision integer. You do not need to read through the code to see how it works, and the README file that was with it was fairly useless. What I would read is the class specification, which starts around line 120 or so. It is easy to read the supported operators, constructors, logicals, and type converters. You will not need to add anything to this list of functionalities, which is fairly extensive. You will note that some of the functions have a comment "throw" beside it. This means that you can turn on exception handling and catch these exceptions, which we will do, so you may want to look over exception handling from COSC 220.

The second package is the tinyfiledialogs code which is in tinyfiledialogs.h and tinyfiledialogs.cpp. The tinyfiledialogs package is a cross-platform package that will easily invoke the system-level standard dialog boxes for file opening and saving, among other things.

Below are some program specifics and after that a long run of the program as an example. You may want to look over the example first and then refer to it as you read through the specifications below.

2.1.1 Program Specifications

1. You can use the same tree node and expression tree data classes as in the class example, with some minor alterations. In the ExpData class you will want to change the num variable from double to InfInt and the op from a char to a string. I would like you to deal with the functions as strings and not use a character correspondence as in the example code. This is a much better design and removes a lot of code with only some minor alterations to the get token function.

2. The BNF for the expression-term-factor-power functions remains the same. Hence the structure (and a good portion of the code) of these functions will remain the same.

3. The input lines are to be input as strings, parsed into a tree and then evaluated. Note that there are spaces in the inputs, so you will need to do getlines in place of streaming.

4. The program should be able to handle the following operations. For `+ - * / %` the operands can be the InfInt numbers but for the power, ^, the left operand is an InfInt type and the right operand is to be a C++ integer. In the parsing of the expression to the tree you can let the right operand be an InfInt but when you evaluate it you will want to convert it to an int before doing the calculation.

   (a) +: addition.

   (b) −: subtraction as well as the unary minus. I would suggest keeping the use of ~ in the tree, this way you know from the symbol whether to evaluate both trees or just the right tree.

   (c) ∗: multiplication.

   (d) /: integer division, note in the example below `4378461647/8594702` returned 509 and not 509.4372843875215. As with any integer division the decimal portion is to be truncated and not rounded.

   (e) %: modulus. Note that the modulus has the same order of precedence as multiplication and division.

   (f) ^: power, only non-negative powers should be accepted.

5. The program should be able to handle the following functions.

   (a) fib: The nth Fibonacci number, ex. fib(1000) should return the exact value of the 1000^{th} Fibonacci number. I started the sequence at index 1, so fib(1) = 1, fib(2) = 1, fib(3) = 2, ...

(b) fact: The factorial of the number, so fact(1000) should return the exact value of 1000!.

(c) sqrt: The integer square root of a number, that is, the integer portion of the square root. Note that sqrt(57849275208574328 57) returned 2405187626 and not 2405187626.955002. Again, a truncation. Although the algorithm to do this is not difficult, it is a built in function in the InfInt class.

6. You can also use any previous input or output in the workspace by using i# for inputs and o# for outputs. For example, i4 * 2345 will take the input number 4 and multiply it by 2345. Similarly, i4 * o7 will multiply input 4 by output 7. Hint: since the main program will be responsible for keeping track of the inputs and outputs it may be easier to preprocess the input string in the main before sending it to the expression tree. That is, replace all of the i# and o# markers in the string with the mathematical expressions from the input or output strings. More specifically, consider the example below.

```
i1 > 2^100
o1 > 1267650600228229401496703205376

i2 > i1 *1234
o2 > 1564280840681635081446931755433984

i3 > i2 / 327842326847
o3 > 4771442588655325758199

i4 > save
Saving Workspace to /home/don/Programming/COSC320Labs/Proj01/test001.iwf

i4 > open
Opening /home/don/Programming/COSC320Labs/Proj01/test001.iwf

i4 > 2^100
o4 > 1267650600228229401496703205376

i5 > (2^100) *1234
o5 > 1564280840681635081446931755433984

i6 > ((2^100) *1234) / 327842326847
o6 > 4771442588655325758199
```

Input #2 would be altered to `(2^100)*1234` before sending it to the parser. Input #3 would be altered to `((2^100)*1234) / 327842326847` before sending it to the parser. As you can see by the save/open sequence this is what was done and saved. This will also save you from having to do replacements of replacements of replacements and so on.

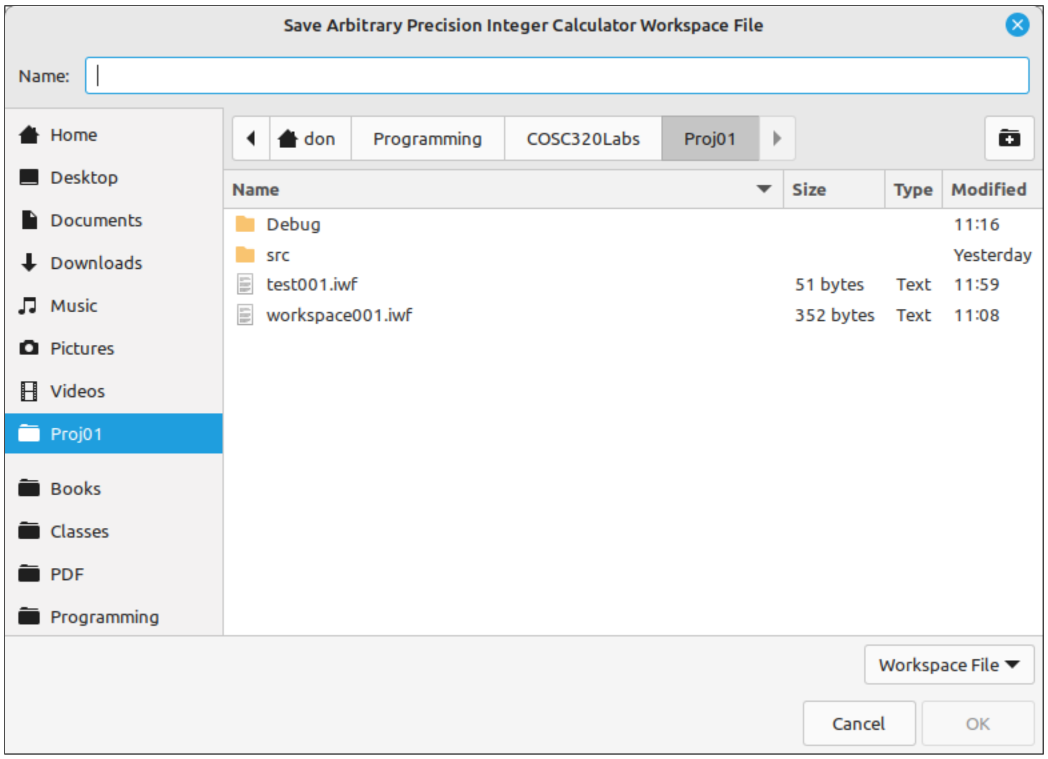
7. An input of 'h' will print out a short help screen on syntax and functions.

8. An input of 'save' will save the current workspace inputs (no outputs) to a text file that the user chooses. For this, you will invoke the tinyfiledialogs save dialog box. The code to do this is below. The file patterns sets the extension of the files to show in the dialog, we will use ".iwf" for our file extensions. Then `tinyfd_saveFileDialog` will show

the save dialog with title "Save Arbitrary Precision Integer Calculator Workspace File", no starting directory (the NULL), 1 is the number of patterns, then the pattern array, then the title of the pattern. You can add patterns if you want but these will be sufficient. The return is a C-style string that is both the filename and path to the file as selected by the user. If the user clicks on the cancel button the return is NULL so that can be checked easily with an if statement.

```
char const *lFilterPatterns[1] = { "*.iwf" };

char *filename = tinyfd_saveFileDialog(
        "Save Arbitrary Precision Integer Calculator Workspace File",
        NULL, 1, lFilterPatterns, "Workspace File");

if (filename) {

// Code to make sure that the .iwf extension is on the filename.
// Code to save the workspace inputs to a text file with name filename.

}
```

When this code is run the following dialog box will appear allowing the user to select a directory and type in a filename.



Name	Size	Type	Modified
Debug			11:16
src			Yesterday
test001.iwf	51 bytes	Text	11:59
workspace001.iwf	352 bytes	Text	11:08

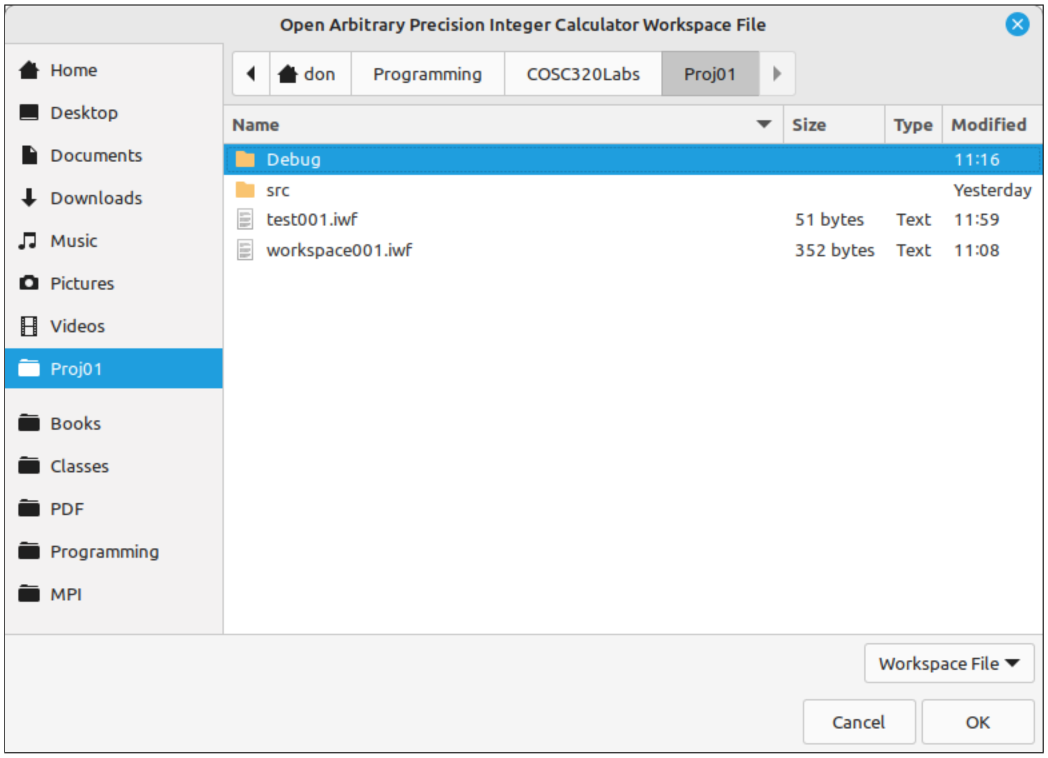
The style and appearance of this dialog depends on the operating system and may look different on Windows, Mac, and different flavors of Linux.

9. An input of 'open' will open a workspace file and load all the inputs in the file into the current workspace as well as evaluate each of them. For this, you will also invoke the tinyfiledialogs open dialog box. The code to do this is below. All the parameters are

the same as the save dialog but the last is a boolean that specifies if multiple selections are allowed, we will not be doing multiple file selections, hence the 0. The return is a C-style string that is both the filename and path to the file as selected by the user.

```
char const *lFilterPatterns[1] = { "*.iwf" };

char *filename = tinyfd_openFileDialog(
        "Open Arbitrary Precision Integer Calculator Workspace File",
        NULL, 1, lFilterPatterns, "Workspace File", 0);

if (filename) {

// Code to load the file into the current workspace.

}
```

When this code is run the following dialog box will appear allowing the user to select a directory and type in a filename.



Name	Size	Type	Modified
Debug			11:16
src			Yesterday
test001.iwf	51 bytes	Text	11:59
workspace001.iwf	352 bytes	Text	11:08

The style and appearance of this dialog depends on the operating system and may look different on Windows, Mac, and different flavors of Linux.

10. Note that on an open or save there is a message that tells the user if a file has been opened or saved. Also note that the open, save, or h inputs are not added to the input or output lists and the i# and o# designations are not incremented.

11. Incorporate error checking into the program by utilizing the exception handling system built into the InfInt class. The example run below does not show errors, here is an example of some of this. The errors may occur in either in parsing the expression or in the evaluation of the expression.

```
i1 > 2^(-3)
o1 > Error: Exponent must be non-negative.

i2 > fact(fib 10))
o2 > Error: Left Parenthesis Expected.

i3 > fact(fib(20)
o3 > Error: Right Parenthesis Expected.

i4 > sin(345)
o4 > Error: Function not found.

i5 > fib(-2)
o5 > Error: Fibonacci value must be non-negative.

i6 > fact(-100)
o6 > Error: Factorial value must be non-negative.

i7 > sqrt(3.25)
o7 > Error: Right Parenthesis Expected.

i8 > sqrt(-5)
o8 > Error: Square root value must be non-negative.

i9 > 2^723695176534176507365923765937456927652
o9 > Error: out of bounds
```

Several suggestions here, first I would store a string in the tree class that holds the first error that is encountered, it is possible that there are several errors are propagated in a single expression. So if a "Right Parenthesis Expected." error happens and then an "out of bounds" error happens the one that is reported is the "Right Parenthesis Expected.".

Another facet to this is that the error could happen in your tree class during parsing or evaluation, or it could happen in the InfInt class if you send it something it cannot do. In the example above, the "Right Parenthesis Expected." (and the left, as well as the function not found) came from the parser, the "Fibonacci value must be non-negative." came from the evaluator. These should be easy to track, just replace the cout errors in the example with code to set the error string of the tree. On the other hand, the "out of bounds" error came from the InfInt class. Recall from above that the for powers you are restricting the power to be of integer size, as you are the fact and fib inputs. So in the last error our evaluate did evaluate the exponent (right hand child of the ^ node) but then asked InfInt to convert it to an integer, this threw an exception which we caught and put the InfInt error string "out of bounds" into our tree's error string.

To set this up do the following, you might want to take a quick look over exception handling from COSC 220. First you need to define `INFINT_USE_EXCEPTIONS`. You can put the following at the top of your main or just add it to the top of the InfInt.h file, inside the guard

```
#ifdef INFINT_USE_EXCEPTIONS
```

Now if there is an error in an InfInt function it will throw an exception instead of printing the error out to the console. Use a standard try-catch statement to catch and process the error. Specifically,

```
try {
...
} catch (InfIntException &e) {
    setError(e.what());
    ...
}
```

12. Since you are moving from storing a character operation to a string, you will need to update the getToken function. In the example I use string searches to determine if a character is valid, a better method would be to use functions like, isalnum, isdigit, isalpha, etc.

13. White space should not be an issue. That is `2*3`, `2  *3`, `2  *  3`, and `2*   3`, should all return 6. The neat thing here is that you do not need to remove the whitespace from the input string if you simply skip over whitespace characters in the getToken function.

14. Finally, an input of exit will end the program.

2.1.2 Example Run

```
Arbitrary Precision Integer Calculator
Standard arithmetic expression evaluator but with unlimited integer size.
by Don Spickler

Type 'H' for help
Type 'exit' to quit.
Type 'open' to open a workspace file.
Type 'save' to save a workspace file.


i1 > 329475173407076 * 43896378269881769417251273
o1 > 14462766842411898748885363523442208207748

i2 > 2^100
o2 > 1267650600228229401496703205376

i3 > 4378461647/8594702
o3 > 509

i4 > 4378461647 % 8594702
o4 > 3758329

i5 > 738402 - 547832907420875
o5 > -547832906682473

i6 > (4738756 - 7589347)^5 + 5381057*62384 - 34789257 % 22222
o6 > -188223804828232656174081357625890

i7 > -2785 + 2789 * 23746877
o7 > 66230037168

i8 > 3^1234
o8 > 5856367529932071269049608726415028439757143623443063785468784843536946634255003８
5879684245345990518684604685177429244575314953213864840914056159382322242855920028381
6367593571556942770711203894947318922509986128995034233177140247221695941679146171215
6751581543586852243090731354814865404274996443634848152189914882304406873590219651150
6767854521236572567824287848592088543692703037258014725677578888237430186684544106507
```

```
6726499613479868210410548366056401445912760529623658838510267592199677099676046090140
514526228797525926977514932394954796457727416398081482661262807288229389463819882569

i9 > fact(100)
o9 > 93326215443944152681699238856266700490715968264381621468592963895217599993229915
608941463976156518286253697920827223758251185210916864000000000000000000000000

i10 > fib(1234)
o10 > 3477467391803702010525174406043359697886849349278437106573522393041216496868459
6797563645939245305337749302687502074476014584240179237874932111371991961858809572448
5583919541019961884523908359133457357334538791778480910430756107407761555218113998374
287548487

i11 > sqrt(5784927520857432857)
o11 > 2405187626

i12 > i7 * 67348256
o12 > 4460477498079979008

i13 > o7 * i4
o13 > 248914269359572272

i14 > i4^10
o14 > 562273934055147239310310400850098096342203246584703513534775977201

i15 > h
=== Arbitrary Precision Integer Calculator Help ===
Type 'H' for help
Type 'exit' to quit.
Type 'open' to open a workspace file.
Type 'save' to save a workspace file.

Supported Operations and Functions:
+: addition, Ex: 123 + 432
-: subtraction, Ex: 123 - 432
*: multiplication, Ex: 123 * 432
/: integer division, Ex: 123 / 432
%: modulus, Ex: 123 % 432
^: power, Ex: 123 ^ 432 --- exponent must be of integer size.
(): parentheses for expression delimitation.
fact: factorial, Ex: fact(1000) --- operand must be of integer size.
fib: Fibonacci number , Ex: fib(1000) --- operand must be of integer size.
sqrt: integer square root, Ex: sqrt(1000)
You can also use previous inputs and outputs by using their designations,
for example, i4 * 2345 will take the input number 4 and multiply it by 2345.
Similarly, i4 * o7 will multiply input 4 by output 7.

i15 > save
Saving Workspace to /home/don/Programming/COSC320Labs/Proj01/workspace001.iwf

i15 > 4 * i11
o15 > 9620750504

i16 > open
Opening /home/don/Programming/COSC320Labs/Proj01/workspace001.iwf

i16 > 329475173407076 * 43896378269881769417251273
o16 > 14462766842411898748885363523442208207748

i17 > 2^100
o17 > 1267650600228229401496703205376

i18 > 4378461647/8594702
o18 > 509
```

```
i19 > 4378461647 % 8594702
o19 > 3758329

i20 > 738402 - 547832907420875
o20 > -547832906682473

i21 > (4738756 - 7589347)^5 + 5381057*62384 - 34789257 % 22222
o21 > -188223804828232656174081357625890

i22 > -2785 + 2789 * 23746877
o22 > 66230037168

i23 > 3^1234
o23 > 585636752993207126904960872641502843975714362344306378546878484353694663425500 3
8587968424534599051868460468517742924457531495321386484091405615938232224285592002838
1636759357155694277071120389494731892250998612899503423317714024722169594167914617121
5675158154358685224309073135481486540427499644363484815218991488230440687359021965115
0676785452123657256782428784859208854369270303725801472567757888237430186684544106 50
7672649961347986821041054836605640144591276052962365883851026759219967709967604609014
0514526228797525926977514932394954796457727416398081482661262807288229389463819882569

i24 > fact(100)
o24 > 9332621544394415268169923885626670049071596826438162146859296389521759999322991
5608941463976156518286253697920827223758251185210916864000000000000000000000000

i25 > fib(1234)
o25 > 3477467391803702010525174406043359697886849349278437106573522393041216496868459
6797563645939245305337749302687502074476014584240179237874932111371991961858809572448
5583919541019961884523908359133457357334538791778480910430756107407761555218113998374
287548487

i26 > sqrt(5784927520857432857)
o26 > 2405187626

i27 > (-2785 + 2789 * 23746877) * 67348256
o27 > 4460477498079979008

i28 > (66230037168) * (4378461647 % 8594702)
o28 > 248914269359572272

i29 > (4378461647 % 8594702)^10
o29 > 562273934055147239310310400850098096342203246584703513534775977201

i30 > exit
```

2.2 Arbitrary Length Rational Number Calculator (Optional)

This exercise is optional for extra credit. Note that this is a substantial update of the previous exercise and will need to be complete for any awarding of extra credit. I do not want to see the code for the last exercise with a couple minor updates. Submissions of that type will not receive any extra points. So either do this seriously, or not at all. As the saying goes, "Go big or go home."

Once finished with the integer calculator, we will expand the program for doing exact arithmetic with rational numbers (fractions) of arbitrary length. For this you will create another class structure, RationalNumber, which will store InfInt data types for the numerator and denominator. The ExpData class will change its num member variable to the new type

RationalNumber and hence the code that operated on InfInt types will be changed to operate on RationalNumber types. Specific updates to the program specifications are below.

2.2.1 Program Specifications

1. First create a RationalNumber class that stores a numerator and a denominator that are both of InfInt type.
   (a) Overload constructors that take RationalNumber objects, InfInt objects, and two InfInt objects (numerator and denominator). Have a default constructor that creates a rational number of 0.
   (b) Overload assignment for a rational number and an InfInt.
   (c) Overload +, -, *, / for two RationalNumber objects, a RationalNumber object and InfInt object, and an InfInt object and a RationalNumber object.
   (d) Overload ˆ for a RationalNumber object and an int.
   (e) Overload +=, -=, *=, and /= for a right operand of a RationalNumber object and a right operand of an InfInt object.
   (f) Overload == for two RationalNumber objects, a RationalNumber object and InfInt object, and an InfInt object and a RationalNumber object.
   (g) Overload the other 5 logical operations for two RationalNumber objects. The other combinations are up to you to overload or not. Note that you cannot just get an approximation for the rational numbers and compare the numbers. The following may help. Note that if the denominators of the rational numbers are positive then the following is true,

$$\begin{aligned}\frac{a}{b} = \frac{c}{d} &\iff a \cdot d = b \cdot c\\ \frac{a}{b} < \frac{c}{d} &\iff a \cdot d < b \cdot c\\ \frac{a}{b} > \frac{c}{d} &\iff a \cdot d > b \cdot c\\ \frac{a}{b} \leq \frac{c}{d} &\iff a \cdot d \leq b \cdot c\\ \frac{a}{b} \geq \frac{c}{d} &\iff a \cdot d \geq b \cdot c\\ \frac{a}{b} \neq \frac{c}{d} &\iff a \cdot d \neq b \cdot c\end{aligned}$$

   The operations on the right will all be with InfInt data types that already have the logical operators overridden.
   (h) Overload the stream out << operator.
   (i) You will probably want some accessor and mutator functions. Some of the functions we are implementing are for integers only so you will probably want a boolean function (say isInt()) that will return true if the rational number is really an integer, that is, its denominator is 1 after the fractions is reduced.

(j) You will want a reduce function that will adjust the fraction to its lowest terms. You do not want an output of 4/8 on the screen, you would want this to be reduced to 1/2. To reduce a fraction you find the GCD (greatest common divisor) between the numerator and the denominator and then divide the GCD into each of the numerator and the denominator. So for example, with 12/15, the GCD is 3, so 12/3 = 4 and 15/3 = 5, hence 12/15 = 4/5. You will also find it convenient if for negative numbers that the numerator is the one that is negative and the denominator is positive. So an algorithm for the reduction function that will adjust the signs, calculate a GCD, and divide would be the following.

Algorithm 1 Reduce: *num* is the numerator and *den* is the denominator.

```
if $den < 0$ then                                  ▷ Adjust signs.
    $num \leftarrow -num$
    $den \leftarrow -den$
end if
if $num = 0$ then                                  ▷ Special case of 0.
    $den \leftarrow 1$
else
    $x \leftarrow num$                             ▷ gcd($num, den$) calculation.
    $y \leftarrow den$
    if $x < 0$ then
        $x \leftarrow -x$
    end if
    while $x \mod y > 0$ do
        $t \leftarrow y$
        $y \leftarrow x \mod y$
        $x \leftarrow t$
    end while                                      ▷ $y = \gcd(num, den)$
    $num \leftarrow num/y$
    $den \leftarrow den/y$
end if
```

You will be calling this function fairly often. Frankly, just about any time you do any arithmetic operation on a rational number it will need to be reduced.

2. Update the tree functions to work with the RationalNumber objects in place of the InfInt objects. Specifically,

   (a) The operations +, -, *, and / will need to work with two RationalNumber objects.

   (b) The ^ operator will need to work with a RationalNumber object on the left and an integer on the right. So this needs to be an integer mathematically and of int size.

   (c) The `%` operator will need to work with two RationalNumber objects that are mathematically integers but need not be restricted to int size. That is, they most both have denominator 1 when reduced and the modulus will be between their numerators which are both InfInt data types.

(d) The fib and fact functions need to work with a RationalNumber object that is an integer mathematically and of int size.

(e) The sqrt function is again an integer square root. So its operand is a RationalNumber object that is mathematically an integer but need not be restricted to int size. That is, have denominator 1 when reduced and the sqrt function will be applied to the numerator, which is an InfInt data type.

(f) Add in another function, invert, that will take the reciprocal of the fraction. So invert(5/9) will return 9/5. Note that this might give you a division by zero, which should turn up as an error and not an out out like 1/0.

3. As with the first exercise, the incorporation of tags i# and o# need to work, as well as file loading and saving, help screen and exit. You should not need to do much, if anything, here.

4. The same error checking on expressions should be incorporated here as well. Note that you will have some new errors if the user tries to apply the fib or fact on a non-integer. Same would be true with other possible errors like a non-integer with the modulus.

2.2.2 Example Run

```
Arbitrary Precision Rational Number Calculator
Standard arithmetic expression evaluator but with unlimited integer size.
by Don Spickler

Type 'H' for help
Type 'exit' to quit.
Type 'open' to open a workspace file.
Type 'save' to save a workspace file.


i1 > (2/3)^50
o1 > 1125899906842624/717897987691852588770249

i2 > invert(57/23)
o2 > 23/57

i3 > 637423472695/4382574097820
o3 > 127484694539/876514819564

i4 > fib(123)
o4 > 22698374052006863956975682

i5 > fact(42
o5 > Error: Right Parenthesis Expected.

i6 > fact(42)
o6 > 1405006117752879898543142606244511569936384000000000

i7 > fact(3.5)
o7 > Error: Right Parenthesis Expected.

i8 > fact(3/5)
o8 > Error: Factorial value must be an integer.

i9 > invert(0/1)
```

```
o9 > Error: Division by 0.

i10 > invert(0)
o10 > Error: Division by 0.

i11 > 2017/344382084
o11 > 2017/344382084

i12 > i1 / i11
o12 > 14360731714587730116608/5362963856201728413146 6379

i13 > i12^2
o13 > 2062306153783658468460362741021920532774256 64
  /2876138132292611311290800292391014543072680807371641

i14 > (1/2) % 17
o14 > Error: Operands must be integers.

i15 > 12345 % 1555
o15 > 1460

i16 > exit
```