Assignment #2:

To complete this assignment, you will email the following files to the TA prior to the deadline. You may work in pairs. We recommend each member of the pair attempt the theory questions separately at first to maximize learning and then come together to write up an answer. At the top of each file state the full names and logins of each student.

- theory.txt – answers to I, II
- code.hs – answers to III, IV, V. This file must type check and compile using Haskell or you will receive a zero for this part of the assignment.

Part I:

Give answers to this part in the theory.txt file. Do your proofs in 2-column style as shown in class with one step per line and a justification in parens beside it. Do not skip any steps. Show each fold or unfold step you do separately. If you need to do an arithmetic manipulation, show that as a step separate from any fold/unfold of a definition. You may use any basic facts about arithmetic you need such as associativity or commutativity of multiplication. Write (simple arithmetic) to justify an arithmetic manipulation step. Assume that all floating point operations have perfect precision (ie: they actually operations on real numbers; there are no rounding errors). Consider the following Haskell type and function definitions:

```
data Shape =
    Rectangle Side Side
  | Ellipse Radius Radius
  | RtTriangle Side Side

type Side   = Float
type Radius = Float

area :: Shape -> Float
area (Rectangle s1 s2)  = s1 * s2
area (Ellipse r1 r2)    = pi * r1 * r2
area (RtTriangle s1 s2) = (s1 * s2) / 2

circleArea :: Radius -> Float
circleArea r = pi * r * r

circle r = Ellipse r r

rectArea :: Side -> Side -> Float
rectArea s1 s2 = area (RtTriangle s1 s2) + area (RtTriangle s2 s1)
```

a) Prove by calculation, not induction, that for all r,
   `circleArea r = area (circle r)`

b) Prove by calculation, not induction, that for all s1 and s2,
   `rectArea s1 s2 = area (Rectangle s1 s2)`

Part II:

Give answers to this part in the theory.txt file.

Consider the following Haskell functions where minbound is the minimum Int and max finds the maximum of two Int.

```
zip :: ([a], [b]) -> [(a, b)]
zip ([ ], _) = [ ]
zip (_, [ ]) = [ ]
zip ((x:xs), (y:ys)) =  (x, y) : zip (xs, ys)

unzip :: [(a, b)] -> ([a], [b])
unzip [ ] = ([ ], [ ])
unzip ((x,y):zs) =
  let (xs,ys) = unzip zs in
  (x:xs, y:ys)

maxs :: [Int] -> Int
maxs [ ] = minbound
maxs (x:xs) = max x (maxs xs)
```

a) Theorem: For all finite lists xs :: [(a,b)], zip (unzip xs) = xs.

   Proof: By induction on the structure of xs. Lay out the proof as you have been instructed to do in class. Points will be given for having the right cases for the proof, stating the appropriate induction hypotheses, laying out the proof in 2-column style with appropriate justifications, etc. It is better to write "I don't know the justification" in your proof than to supply a wrong justification.

b) Falsehood: For all finite lists xs :: [a] and ys :: [b], unzip (zip (xs, ys)) == (xs, ys).

   Disprove this statement by giving a counter-example. In other words, given an example of an xs and a ys for which the statement above is false.

c) Theorem: For all finite lists xs :: [Int] and ys :: [Int], max (maxs xs) (maxs ys) == maxs (xs ++ ys).

   Proof: By induction on the structure of xs. You may use any obvious property of max and minbound that you like (be explicit about the properties you use in the justifications in your proof). For example, you may use commutativity, associativity, or the fact minbound is a minimum:

   max a b = max b a
   max (max a b) c = max a (max b c)
   max a minbound = a

d) Conjecture: For all lists xs :: [a], length (3:xs) > length xs.

   Either prove it or give a counter example in Haskell.

Part III

In the first week of lecture, we gave mathematical definitions of the syntax binary and unary numbers. Those mathematical definitions looked like this:

b ::= # | b0 | b1

i ::= # | iS

Boy, that sure looks a lot like a Haskell data type doesn't it? That is not a coincidence! In this question, you will implement those mathematical definitions as directly as possible in Haskell. In the file code.hs, implement the following types and functions:

```
data Bin = …
data  Un = …

-- the denotational semantics of a binary number as an Int
binSem :: Bin -> Int

-- the denotational semantics of a unary number as an Int
unSem  :: Un -> Int

-- convert a binary number to a unary number
bin2un :: Bin -> Un

-- convert a unary number to a binary number
un2bin :: Un -> Bin
```

Your functions should satisfy the following properties. Do not hand in proofs of these properties. However, doing these proofs is good prep for exams.

Theorem: for all u::Un, binSem (un2bin u) = unSem u

Theorem: for all b::Bin, unSem (bin2un b) = binSem b

Think about, but do not hand in:

Conjecture 1: for all b::Bin, un2bin (bin2un b) = b.

Conjecture 2: for all u::Un, bin2un (un2bin u) = u.

Conjecture 3: for all b1,b2::Bin, binsem b1 = binsem b2 implies b1 = b2

Which of these conjectures are true for your definitions? Is the analogue of conjecture 3 true for unary numbers? What does conjecture 3 lead you to believe about the relative "goodness" of your implementation of binary numbers? Feel free to discuss these questions with your classmates or with the professor or TA during office hours.

Part IV: Practice with Higher-order Programming

In the file code.hs, define the following functions. When asked for a "recursive" version of the function in question, you should turn in an explicitly recursive function that calls itself:

```
foo x = … foo x …
```

When asked for a "non-recursive" function, you should not yourself write a function that calls itself. Instead, you should use `map`, `fold`, function composition and other standard functions from the Haskell prelude to construct your function. Explicitly give the type of each function.

a) `pairAndOneRec` – a recursive function that for each Int element of a list, pairs that list element with itself plus one. Eg: pairAndOneRec [0, 5, 10] = [(0,1), (5,6), (10,11)]

b) `pairAndOneNon` – as above but nonrecursive

c) `addPWRec` – given a list of lists of Int. Return a list of Int in which the ith element of the resulting list is the sum of the ith elements of the argument lists. This is often referred to as "pointwise" application of the addition function The lists do not have to be the same length: Eg: `addPWRec [[1, 5, 9], [2, 3], [7], [2, 2]] = [12, 10, 9]`

d) `addPWNon` – as above but nonrecursive.

e) `minListRec` – a recursive function: return the minimum positive element or zero if there is no positive element. Eg: minListRec [1, -3, 4] = 1

f) `minListNon` – as above but nonrecursive

Part V : Drawing Fractals

For this assignment we will have to set up some graphics libraries for use with Haskell. We will also be using code from Paul Hudak's "Haskell School of Expression" (SOE). To install the graphics library and SOE, see the course web pages. Test that it is working by going to the SOE/src directory and typing

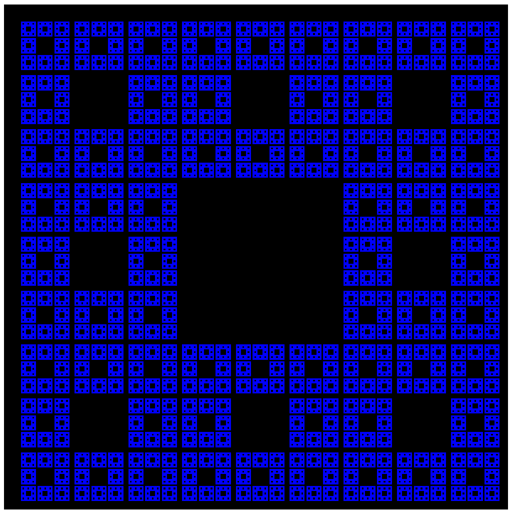
```
$ ghci Draw.lhs

*Draw> main0
```

You should see a graphics window displayed with a couple of shapes in it.

Read the handout on using Haskell graphics and Sierpinski Triangles.

a) In the file code.hs, implement a function called "carpet" (with well-structured auxiliary definitions) that brings up a window displaying the following picture. A keypress by the user should close the window. You will either need to run your program in SOE/src or add that path to GHC's search path. You will need to put "import SOE" at the top of your file.



b) In the file code.hs, implement a function "fractal" (again with well-structured auxiliary definitions) that brings up a window displaying a fractal pattern of your own design. A key press by a user should close the window safely. Be creative! The only constraint is that it shows some interesting self-similar recursive pattern. To get additional ideas, try Google or Wikipedia for information on fractals. Explain your pattern and any mathematics behind it in a comment preceding your fractal function. As always, cite sources for your ideas. Grading for this part of the assignment will be done in 4 categories: (0) Not handed in or doesn't compile/draw, (1) Minor variation of Sierpinski triangle/carpet, (2) Major departure from Sierpinski triangle/carpet demonstrating significant thought, (3) Outstanding, sophisticated, surprising, creative, artistic, detailed.