Suppose we take an expression in $x$ and $y$, built by nesting simple primitives:

$$
\ldots \times \ldots \\
\text{average}(\ldots, \ldots) \\
\sin(\pi \times \ldots) \\
\cos(\pi \times \ldots).
$$

If we compute the value at any point with $-1 \leq x, y \leq 1$, the result will also be between $-1$ and 1. By scaling the answer to a grayscale value 0–255, we can plot the function in this 2-by-2 square. (From three such expressions, we can get red, green, and blue values for each point.)

Surprisingly, every sufficiently complicated expression has a decent chance of being interesting! We can generate deeply-nested expressions completely randomly, see how they look, and keep the good ones.

**Preliminaries**

You may work on this assignment with a partner (preferred) or by yourself. If you work with a partner, you must follow pair programming rules (see the [PairProgramming](https://wiki.cs.hmc.edu/index.php?title=PairProgramming) page on the wiki for full rules). Both members of the pair must fully understand all jointly submitted work. You are not required to work with your partner from lab. You must use the same partner for Assignment 3a and 3b.

Files for this assignment are at [https://svn.cs.hmc.edu/cs131/spring16/given/hw3](https://svn.cs.hmc.edu/cs131/spring16/given/hw3). Copy them in the usual way (see [CopyingAssignmentFiles](https://wiki.cs.hmc.edu/index.php?title=CopyingAssignmentFiles) on the wiki for a reminder of how).
Required Code Quality

The same rules apply as in the last homework. Write clear code, and appropriate comments. You code must compile, type check, use the function names specified in the assignment, etc.

Random Art

The code in Randomart.hs has a mostly-complete implementation of a random art generator. (Warning: the supplied code relies on the convert command-line tool from the ImageMagick suite. This program exists on knuth and all other HMC CS machines.)

1. The code starts by defining some types to represent expressions and points:

```haskell
data Exp = X  -- x  
   | Y  -- y
   | SinPi Exp  -- sin (pi * e)
   | CosPi Exp  -- cos (pi * e)
   | Avg Exp Exp  -- average of e1 and e2
   | Times Exp Exp  -- product of e1 and e2

deriving (Show)
```

```haskell
type Point = (Float, Float)
```

Complete the definition of the function

```haskell
eval :: Exp -> Point -> Float
```

that evaluates the given expression at the given \((x, y)\) coordinates.

- **Tip:** When running ghci, it’ll run your code much quicker if you invoke it as

  ```bash
  ghci -fobject-code
  ```

  because that will cause it to *compile* the code you use in the interactive session rather than *interpret* it. This assignment is quite compute intensive, so you’ll save a lot of time if you remember to run ghci this way.

2. Test your code under ghci by creating a picture of the expression `sampleExp` by running `toPNGgray "test" 300 (eval sampleExp)`. Check the resulting image in your favorite image viewer. (It should look like the image on the first page of this assignment!)

3. Compile the code into an executable using ghc, as explained in the comments at the top of the file. Verify that you get a really boring picture when you run the program from the command line.
4. The function

\[
\text{build :: Int} \to \text{RandomFloats} \to \text{Exp}
\]

is supposed to generate interesting random expressions, but doesn’t; fix it.

The first parameter to build is a maximum nesting depth that the resulting expression should have, and the second parameter will be an infinite list of random numbers between 0.0 and 1.0. (A bound on how deeply you can nest operations (sines, averages, etc.) keeps the expression to a manageable size; it’s easy to write a naïve expression generator that generates incredibly enormous expressions.) A depth of 6 to 11 is a good starting point for experimentation.

Now, if every sort of expression can occur with equal probability at any point, it is likely that the random expression you get will be either \( X \) or \( Y \) (2 chances in 6), or something small like \( \text{Times} \ X \ Y \). Because small expressions produce boring pictures, you should find some way to prevent or discourage this.

**NOTE:** The helper function splitRandomFloats will be useful when implementing build.

5. The provided code allows you to create visualizations of what the existing functions do. From within GHCI, run

- plotOneArg SinPi
- plotOneArg CosPi
- plotTwoArg Avg
- plotTwoArg Times

and examine the resulting PNG files in your favorite image viewer. There is also a plotThreeArg function, but currently no such operations exist in the Exp type.

6. Add at least three new sorts of expressions to the Exp type, and update the build and eval functions appropriately.

   (a) Each new expression form must return a value in the range \([-1.0, 1.0]\) when its arguments are within the range \([-1.0, 1.0]\). If your function goes outside this range, an error will occur during output. Note that floating point math can be imprecise, so that even if in theory your expression should not fall outside the \([-1.0, 1.0]\) range, in practice (depending on the particular operations you use) it might.

   The visualization operations mentioned in the Q5 provide a good check that your function actually works as you might expect over its entire range and does what you’d hope.

   (b) At least one form must involve three Exp subexpressions.

   (c) At least three of your new expression forms must be creative:

   - Simple variants of existing operators (e.g., \((...+...+...)\)/3 or \(\sin(2\pi \times ...)\)) are not creative.
• Functions that can already be built out of existing functions (e.g., \(... \times \times \times \ldots\)) are not creative.
• Applying built-in math functions (e.g. abs, sqrt, atan2) is not terribly creative.

The above does not mean that your functions should be complex (in fact, complexity is detrimental to speed). Rather, you should have some hope that your functions will “do something interesting” from the perspective of random art. There are a number of simple ideas that we will consider creative, given suitable justification in your comments.

The best way to know whether your functions do something useful is to see how much of a difference they make to the pictures your program produces, which leads us to the next part.

7. When you think everything is working and you have added good functions, generate a lot of pictures with different seeds and depths. It is likely that the majority of your pictures will seem mediocre, but hopefully some will be satisfactory. It’s not uncommon to look through one hundred pictures and think 90 of them are pretty terrible, nine are okay, and one seems pretty neat.

One way to generate a lot of pictures is to use ghci -fobject-code (so it’ll run more quickly!), and then start ghci and run something like the following your interactive session.

```
:load Randomart
```

And then, to test the basics in grayscale, run

```
sequence_ [doGray 300 seed depth | seed <- [100..120], depth <- [4..9]]
```

which will produce 126 grayscale images, which should be enough to get a sense of how how interesting your images are. (Remember, you don’t have to wait until the generation is done to start looking at your results.)

If you think these images are okay, you can try generating a large number of color images (252), by running

```
sequence_ [doColor 300 s d1 d2 | s <- [100..120], d1 <- [5..8], d2 <- [1..3]]
```

Depending on your time and inclination, you may want to generate even more pictures or tweak the functions you provide.

8. Once you’re getting random pictures, pick exactly one (per person, pairs submit two), rename it to your knuth username followed by the parameters that generated it, and upload it the RandomArtGallery page of the course wiki.

**Concluding Thoughts**

After finishing you might want to look at [www.random-art.org](http://www.random-art.org) for a more sophisticated implementation that represents expressions as directed graphs rather than just trees, and some pretty cool pictures.