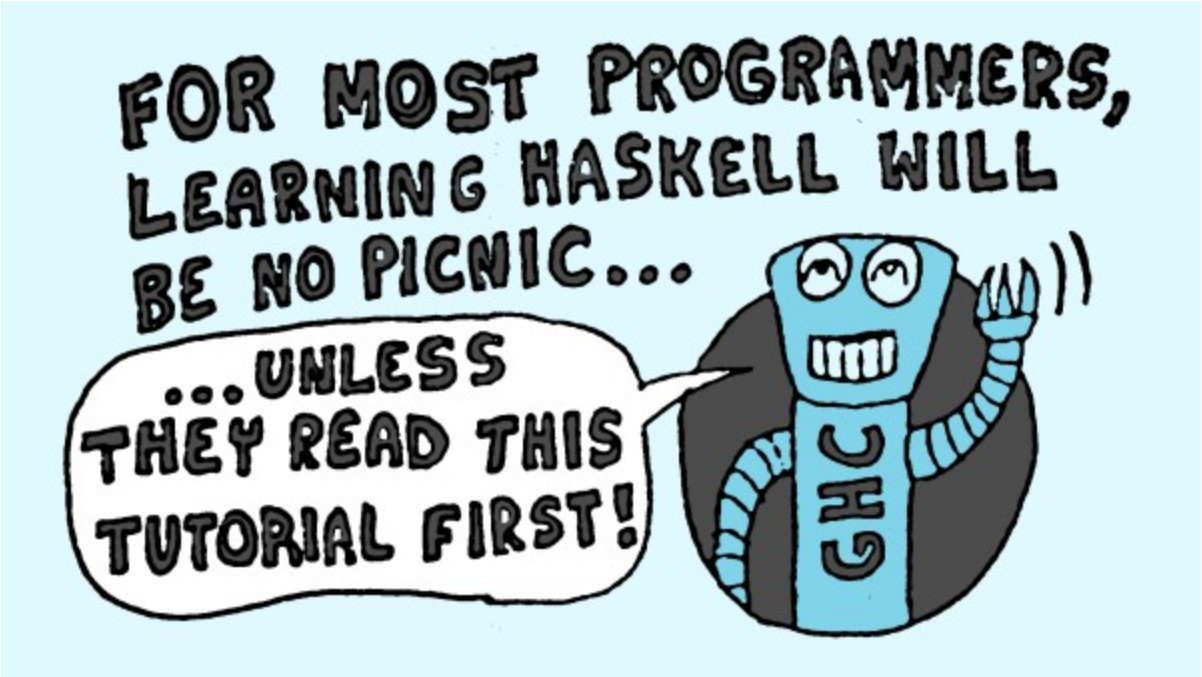


by Conrad Barski, M.D. Mail

Note To Advanced Haskellers

Hi everyone, welcome to my Haskell tutorial!

There's other tutorials out there, but you'll like this one the best *for sure*: You can just cut and paste the code from this tutorial bit by bit, and in the process, your new program will create magically create more and more cool graphics along the way... The final program will have less than 100 lines of Haskell[1] and will organize a mass picnic in an arbitrarily-shaped public park map and will print pretty pictures showing where everyone should sit! (Here's what the final product will look like, if you're curious...)

The code in this tutorial is a simplified version of the code I'm using to organize flash mob picnics for my art project, picnicmob.org... Be sure to check out the site and sign up if you live in one of the cities we're starting off with :-)

NEXT

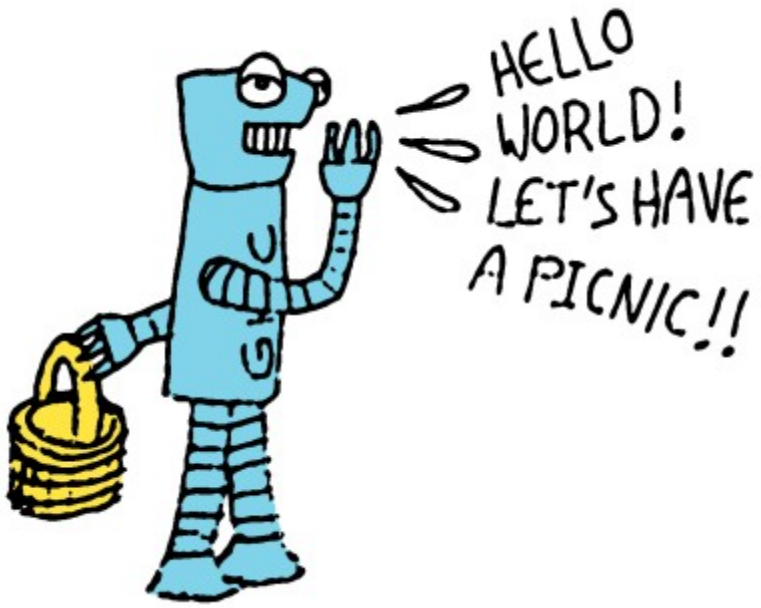
[1] - *Lines of active code only, not counting optional function signatures. Intermediate debug output lines excluded. May cause eye irritation if viewed for prolonged periods of time. Speak to your physician to find out if this tutorial is the best treatment option for you.*

How To Organize a Picnic on a Computer

Ok, so here's what the program is going to do... On the picnicmob.org website, people answer a bunch of random questions. What we want our program to do is take a picture of a city park, a list of answers people gave on the questions, and have it figure out where people should sit in the park so that their neighbors are as similar as possible to them, based on similar answers for the questions. We're going to do this using a standard simulated annealing algorithm. (Read this for more info... we'll be describing the basics below, too...)

Installing GHC Haskell On Your Computer

The only preparation you need to do for this tutorial is install the Glasgow Haskell Compiler- You can get the latest version from here. It's a very well supported and very fast compiler. We're just going to use the basic GHC libraries in this tutorial- No other installs are needed.



Hello World! Let's Have a Picnic!

Now you have all you need to run the "Hello World" program below- Just copy the code into a file called *tutorial.hs*:

```
import Data.List
import Text.Regex
import System.Random
import Data.Ord

type Point     = (Float,Float)
type Color     = (Int,Int,Int)
type Polygon   = [Point]
type Person    = [Int]
type Link      = [Point]
type Placement = [(Point,Person)]

type EnergyFunction a              = a -> Int
type TemperatureFunction           = Int -> Int -> Float
type TransitionProbabilityFunction = Int -> Int -> Float -> Float
type MotionFunction a              = StdGen -> a -> (StdGen,a)

main = do
  putStr "Hello World! Let's have a picnic! \n"
```

After you've copied the code to a file, just run your new program from the command line as shown below:

```
> runHaskell tutorial.hs
```

```
Hello World! Let's have a picnic!
```

phew That was easy!

How Does This Code Work?

As you can see, there's lots of stuff in this "Hello World" that doesn't seem necessary... Actually, only the last two lines (starting at `main = do`) are really needed to print our message... The rest is just header stuff that we'll use later in the tutorial...

At the top, you can see some things we're *importing* out of the standard GHC libraries:

1. The `Data.List` module has extra functions for slicing and dicing lists- Lists are fundamental in Haskell programming and we'll be using them a lot in this program.
2. The `Text.Regex` module let's us do funky things with text strings using *regular expressions*- If you don't know what this means, read this tutorial first. Every programmer should know about regular expressions.
3. The `System.Random` module, as you've probably guessed, lets us use random numbers, essential for simulated annealing.
4. The `Data.Ord` just contains a single function I'm really fond of, called `comparing`. I'm not happy if I can't use `comparing`, and I think we all want this to be a *happy* tutorial...

The next few lines define different types that we're going to use in our picnic program... Haskell programmers are really fond of lots of types, because Haskell compilers are basically just these crazy type-crunching machines that will take any types you give them and if they find anything questionable about how you're using types they will let you know early, preventing your program from becoming buggy. The way Haskell compilers handle types puts them head and shoulders above most other compilers (Read more about Hindley-Milner type inference here.)
The first few `type` lines should be pretty self-explanatory,

for a program that's going to work with pictures: A `Point` is just a pair of floating numbers, a `Color` is an rgb triple, a `Polygon` is just a list of `Points`, a `Person` is just a list of the answers they gave in the questionaire- Very simple stuff...

The lower `type` lines are a bit more difficult to understand- They describe the *types of the four functions* that are used in all simulated annealing algorithms... we'll be describing those later on. For now, just know that in Haskell you want to look at the last arrow, which looks like this `->`... Anything to the left of that arrow is a parameter into our function- The thing to the right is the return value of the function. For instance, the `EnergyFunction` is a function that takes an arbitrary type `a` (which will be a seating arrangement of picnickers) and returns an integer, which is the "energy" of our picnic. A low energy will mean that all the picnickers are happy with where they're sitting. We'll explain these funcitons more later.

What's Good about this Code?

What's good is that in Haskell we can create all kinds of types to describe what we want to do and the Haskell compiler is very good at catching potential errors for us ahead of time. Defining types in Haskell is usually a *choice* and not a *necessity*- This program would run just fine, even if all the type declarations were missing... The Haskell compiler would just figure them out on its own!

Another thing that's good is that the part that writes "Hello World" is very short and simple in Haskell.

What's Bad about this Code?

To keep things simple I'm building these new types using the `type` command, instead of another command called `data`. This other command (which you don't need to know about today) lets you attach a special new type name that you use to construct your new type- That name makes it *even harder* to accidentally use the wrong type in the wrong place in your program.

Another thing thats "bad" about this code is that we used the type `Int`. Unlike all other types in this program, `Int` has an upper limit and could theoretically cause our program to err out if any integers got way too massively big. In Haskell, this type of integer can't get bigger than 2^31... We could have used the type `Integer` instead- This type of integer grows "magically" and can hold *any sized integer*, no matter how big... but it's slower, so we used `Int` instead, since our numbers will never get that big.

NEXT

Alright- So what do we know about the people at our picnic?

Now that we've got "Hello World" working... let's load the answers people gave to the questions on the picnicmob.org website... For testing purposes, I've created some anonymized data and put it into the file people.txt for you to use- Just save this file to your computer.

Here's what that file looks like- It's just a list of list of numbers. Each row is a person and the numbers are the value answer number they gave on each question:

```
[
[2,3,3,4,4,3,5,2,2,3,2,2,2,3,2,5,3,1,3,5,2,5,2,2,2,3,2,5,2,3],
[2,3,3,2,3,3,5,2,3,4,2,2,1,1,1,2,1,5,1,4,2,5,2,2,2,2,4,1,1,1],
[2,3,4,3,3,5,5,2,3,5,2,3,2,1,2,4,4,3,3,1,2,5,3,5,2,3,4,1,1,2],
[1,3,3,3,3,3,3,3,3,4,4,3,3,4,4,2,3,3,3,1,4,3,4,3,3,2,1,3,2,3,3],
[4,1,3,3,3,3,4,1,3,4,3,3,1,2,1,4,4,2,2,4,2,2,2,1,2,3,4,2,5,1],
[2,1,1,2,3,5,4,1,1,1,2,3,5,1,3,2,4,2,3,5,2,5,2,2,2,3,4,2,2,4],
```

```
[1,3,4,2,3,5,4,1,3,1,2,2,2,3,1,1,1,2,2,2,2,1,2,1,2,3,3,1,3,3],
[3,3,3,1,3,2,2,1,2,3,2,3,2,3,2,3,3,5,2,1,2,5,2,1,2,1,4,2,2,2],
[1,3,4,3,3,5,5,3,1,4,2,4,1,1,5,1,1,4,2,5,2,5,3,1,2,3,4,2,1,4],
[1,3,4,2,4,5,4,3,2,1,3,4,1,1,4,5,4,5,3,4,2,2,2,1,2,3,3,4,1,4],
[1,1,1,4,4,1,4,3,3,3,4,3,1,2,1,4,1,5,2,2,2,4,2,5,2,1,4,4,1,3],
...
```

Here's the code that reads in this list of numbers. Adding this code to our existing program is super easy: **Just paste this new code to the bottom of the "Hello World" program to make the new code work-** ***Do the same thing, as we go along, with all the other code fragments in this tutorial.***

```
  people_text <- readFile "people.txt"

  let people :: [Person]
      people = read people_text

  putStr "Number of people coming: "
  print (length people)
```

Now let's see if this code is able to read our file:

```
> runHaskell tutorial.hs

Hello World! Let's have a picnic!
Number of people coming:
200
```

perfect!!

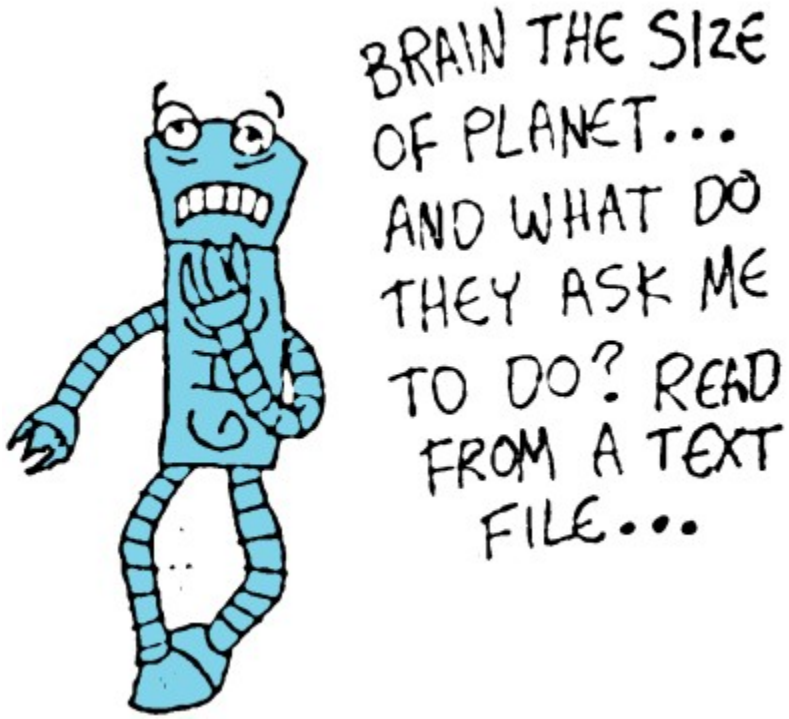
How Does This Code Work?

Ok, so let's dissect this code to figure out how it worked... The first line read the file "people.txt" and assigned it to the a string named `people_text`... Note that we used a left-pointing arrow <− to do the assignment, instead of an equals sign. The reason for this is that Haskell is very finicky about code that does I/O, like reading from the disk or printing something on

the screen. In the Haskell way of thinking, code that does this is *pure evil* and needs to be imprisoned into its own pen, separately from the rest of your Haskell program.

Why does it think I/O is evil? It has to do with those types we defined at the top of our program: For instance, we said that the definition of an `EnergyFunction` is a function that converts from an arbitrary type `a` to an `Int`- *That is all that an `EnergyFunction` is allowed to "do".* In Haskell's way of thinking, if an `EneregyFunction`, let's say, prints something on the screen, then that means it's *doing more than we said it would do*, and will complain when we try to compile our program.

If we want to "break the rules" of Haskell and do stuff like read from the disk or write to the screen, we have to do it in a special way: We put the word `do` at the beginning of the function (remember the line "`main = do`"?) and whenever we grab some value using IO, we use the aforementioned `<-` arrow and it will let us do what we want.



*****IMPORTANT: Before starting to read this next paragraph, put your fingers in both ears and chant "Lalalalala I can't hear you I can't hear you**

lalalala..."***

What Haskell is doing here is forcing you to write functions using IO within the `IO` Monad, which is a concept taken from mathematics and is actually very clever and useful once you learn more about the Haskell way of thinking. ***This is the only time in this tutorial where I'm going to use the word 'Monad'... except for the very very last word of the tutorial****.*

*****OK, you can take your fingers out of your ears and stop chanting now. You can thank me later by email for protecting you from a tragic downward spiral in your sanity that would be inevitable as you try to fully understand the meaning continued in the concepts mentioned in the aforementioned paragraph.*****

So once we've read in our list of people, we want to convert it from a string into a type `[Person]`- Remember, we defined the `Person` type earlier. As you can see, we assign the name `people`, which has the type `Person`, as indicated by the line with the double colon `::`, and `read` it out of the text string `people_text`. Note that in this case, we assigned the name with just a regular 'ol equal sign: We could do this, since we're not doing any IO any more... we already did this "dirty work" earlier. Functions in Haskell that don't do IO are called *purely functional*.

The last two lines are obvious- They just print a message on the screen indicating the length of the `people` variable.

What's Good about this Code?

Several things- First, we were able to decouple our IO from the rest of our code- This helps reduce programming errors and is a major feature of Haskell programming (Clearly, this'll become more important later on, when our code is a bit more complex...)

Something else that's really "good" is that when we read our

`people` from the file, we didn't have to tell Haskell what type of data we were reading: The `read` function in the program used the compiler to figure out, on it's own, that a `Person` is a list of integers, and that reading a list of people therefore constitutes reading a list of list of integers, which is what we put into `people.txt`. If we had put anything else in there, `read` would have noticed this and complained when we ran our program. (In OOP-speak, this basically meant we polymorphically dispatched based on the return value of a function- *Try doing that with your favorite Java compiler sometime* :-)

What's Bad about this Code?

For one thing, we defined our `people` in a `let` expression in the middle of our `main` function... We did this so that we can simplify this tutorial by just appending new bits of code to the bottom of our program as we move along... Most Haskellers would just add these definitions separately outside of the `main` function, but either way works just fine.

NEXT

Let's Draw Us a Purty Picture

Ok, so I promised you some pretty pictures in this tutorial... Now, finally, the time has come for this! We're going to draw pictures in a really cool way: We're going to write our own SVG files from scratch!

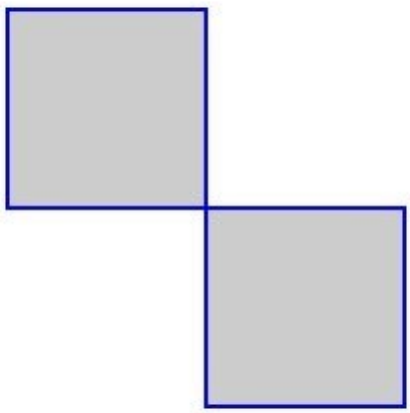
SVG files are a simple file format for specifying images that you'll probably hear a lot more of in the future- It's completely resolution independent and already used by many drawing programs such as Inkscape, which I highly recommend.

(Note to internet explorer users: If the pictures below don't show up right in your web browser, you probably need to download an SVG viewer... everyone else should be fine...)

Here's the code that'll take a list of colored polygons and write it to an svg file:

```
  let writePoint :: Point -> String
      writePoint (x,y) = (show x)++","++(show y)++" "

  let writePolygon :: (Color,Polygon) -> String
      writePolygon ((r,g,b),p) = "<polygon points=\""++(concatMap writePoint p)+
+"\" style=\"fill:#cccccc;stroke:rgb("++(show r)++","++(show g)++","++(show b)+
+");stroke-width:2\"/>"

  let writePolygons :: [(Color,Polygon)] -> String
      writePolygons p = "<svg xmlns=\"http://www.w3.org/2000/svg\">"++(concatMap
writePolygon p)++"</svg>"

  let colorize :: Color -> [Polygon] -> [(Color,Polygon)]
      colorize = zip.repeat

  let rainbow@[red,green,blue,yellow,purple,teal] = map colorize [(255,0,0),
(0,255,0),(0,0,255),(255,255,0),(255,0,255),(0,255,255)]

  writeFile "tut0.svg" $ writePolygons (blue [[(100,100),(200,100),(200,200),
(100,200)],[(200,200),(300,200),(300,300),(200,300)]])
```

After you run this code, you'll see the following picture drawn in the file `tut0.svg` that you can open in your web browser:



How Does This Code Work?

This code is a quick n' dirty SVG file writer. In order to draw a list of polygons, it breaks the task into subtasks- First, into writing the separate polygons, then into the task of writing each individual point.

Since the `Polygons` are just a list of individual `Polygons` and those are just a list of points, we express this in Haskell using the `map` function- This function takes a list of things and a function that operates on an individual item in the list, and *grinds* through the list with this function- Haskellers never use loops- Instead, they either use recursion to do looping, or they use functions like `map` that take other functions as parameters. Functions that do this are called *higher order functions*. In the `writePolygons` and `writePolygon` functions, you can see the mapping function being used: In this example, we're using a clever variant called `concatMap` that also concatenates the result of the mapping together, saving us a step.

An important thing to note in the `writePoint` function is that something funky is happening on the left side of the equals sign- Instead of taking a variable name for the point we're passing in, we're instead pulling in the point using `(x,y)`, so that we can tease the point apart as separate `x` and `y` values- Haskell can do this for us because it supports pattern matching: We can put an arbitrary structure where other languages may ask for a simple variable name- The compiler then figures out if the value "coming in" to the function can meet your structure, and if so, it will *destructure* the parameter and give you whatever pieces you want with the names you gave it.

The way we handle colors in this example uses some more clever Haskell tricks... It has enough complicated ideas that it deserves it's own section in this tutorial: **If you can understand what `colorize = zip.repeat` means, you'll understand probably most of what Haskell has to offer in just three words!**

The Insane coolness of `colorize = zip.repeat`

OK, to explain how the definition of `colorize` could possibly work and why it's so cool, I'll first tell you that it is equivalent to the following, less elegant piece of code:

```
colorize c p = zip (repeat c) p
```

...this version is a little easier to grok for a newbie, so let's start with it first...

To start off, I should point one little oddity about Haskell: It's a lazy language- This means that when your program runs, only those things are calculated at the *last possible moment*- This means Haskell programs can (and usually do) have data structures in them that are actually infinite in size... `repeat` is one example of a function that does this:

Alright- so we take a color, `c`, and use the `repeat` function to turn it into a list of that color that repeats infinitely. So if, for instance, our color is rgb red `(255,0,0)`, the repeat function creates the infinite list [(255,0,0), (255,0,0), (255,0,0), ...]. The fact that this list is infinite is OK in Haskell, because Haskell is just so , well, *damn lazy*- We're not going to ask for all the items in this list, so it'll just happily play around with this infinitely long list as if nothing is wrong.

The next function, `zip`, takes two list and zipps 'em up into a list of pairs: So it'll take our list of colors and our list of polygons in the variable `p` and make a list of `(Color,Polygon)` pairs, which is what we want- Since the number of polygons will probably be finite in number, the zip function will stop using items from the infinite list of colors and, voila, the infinities magically disappear, through sheer, unmitigated, abominable laziness.

Ok, so now let's take this function and let's see if we can make it more elegant

```
colorize c p = zip (repeat c) p
```

To understand how this function can be made even simpler, we need to talk about currying: In Haskell, basically all functions are curried- What this means is that Haskell *functions only ever take a single parameter*. "But wait", you protest, "Our colorize function takes *two* parameters, a `Color` and a `Polygon`." Alas, it is not actually so- To understand why, let's look at the type definition for our

`colorize` function:

```
colorize :: Color -> [Polygon] -> [(Color,Polygon)]
```

Earlier I told you that you should look at the last arrow in a type definition and that all things in front of that arrow are parameters- While this is a good way to think about Haskell functions, in reality, the arrow `->` is actually right associative... that means this function actually reads as follows:

```
colorize :: Color -> ([Polygon] -> [(Color,Polygon)])
```

So in reality, strangely enough, the `colorize` function only takes *one* parameter, but it actually returns another function (of type `[Polygon] -> [(Color,Polygon)]`) as a result! So if we ever write `colorize my_favorite_color my_favorite_polygon` what's really happening is that the color is passed in first and a new function is created- Next, the polygon is passed into this *new* function to generate the result.

OK, so how does this fact help us simplify our definition of `colorize`? Well, remember that Haskell supports pattern matching- This means, we can just drop the `p` (polygon) parameter from both sides and the compiler cna still make sense of it:

```
colorize c = zip (repeat c)
```

Next, we can use a the haskell sequencing operator (represented by the dollar sign `$`) to get rid of the parenthesis- When you see this, it just means there's an imaginary "matching parenthesis" to the dollar sign at the end of the code expression:

```
colorize c = zip $ repeat c
```

The dollar sign is really just *syntactic sugar* but it's really useful because if we are just feeding single values into functions, as we usually are, then it forms kind of an assembly line- So basically, we're taking the value `c`, running it through `repeat`, then running it through `zip`- This works

exactly the same way as a Unix pipe, if you've ever worked with that clever concoction before...

When we see this kind of *assembly line*, we can usually simplify it even more by using function composition, which Haskell represents with a period (You may have learned this in precalculus- Maybe you remember your teacher talking about "f of g of x" in front of the class and putting a little dot between the f and the g- This was exactly the same idea :-)

Using function composition, we can now compose the the `zip` and `repeat` functions like so:

```
colorize c = (zip.repeat) c
```

Can you figure out why we want to do this? You may notice we have another value *dangling off the back* of our function again- We can snip it off of both sides, just like before, leaving us with the final, most elegant form:

```
colorize = zip.repeat
```

You've gotta admit, a language that can do that is pretty cool! Writing functions with the "dangling values" removed is called *writing in a point-free style*

Making Some Simpler Functions For Coloring Stuff

Most of the time, a general-purpose function like `colorize` is too much trouble- What we really want is functions for coloring things red, green, blue, etc. The last thing we defined in the code fragment above were some simple functions with obvious names that are just versions of `colorize` that have the color value prefilled- We do this by just `map`ping a list of rgb values against the `colorize` to create a list of color functions that we just use to define a slew of new functions all at once- Having a language that treats functions just like any other values makes stuff like this easy. As a bonus, this line of code also defines a variable `rainbow` that contains all the colors, and which we'll use later to paint with lots of colors, all at one.

What's good about this code?

One thing that's really good about it is that it does an immense amount of stuff using just a miniscule amount of code- Our `colorize` function is the ultimate in brevity (Remember that all the type definition lines, which have a double colon, are just optional niceties and aren't actually needed for the program to work...)

Why do we care so much about brevity? Well, since Haskell also has ridiculously strict type checking, it means we can sandwich all our bugs between the type checker on one side (many bugs involve mismatching types) and code brevity on the other end (less code, means less room for bugs to hide)

Arguably, Haskell is the one language that has the toughest type checking and allows for the briefest code- That means Haskell allows you write code that's more bug free than almost any other language. I would estimate that I spent only about 3% of my development time debugging this program, which I could never accomplish that in another language. *(Now, mind you, it can be a b*** to get your program to compile properly in Haskell in the first place, but once you get to that point, you're home free :-)*

What's bad about this code?

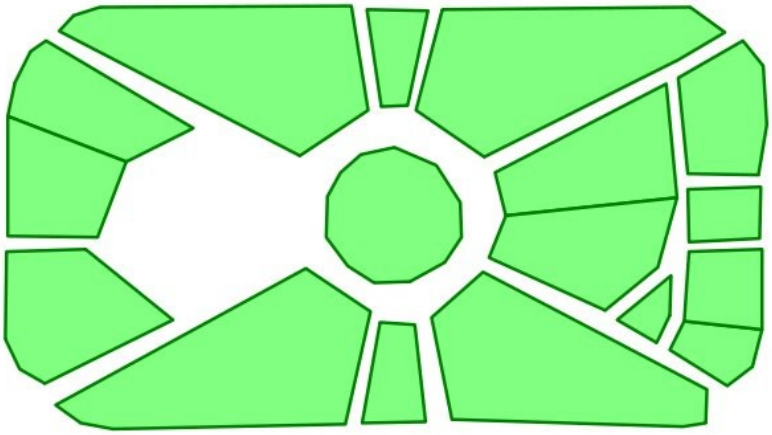
Well, we're creating raw xml data as text in this example- It would be much better to use an XML processing library, like HaXml, to make this even less error prone. However, we want to just stick with the basic GHC compiler here and not

spend time installing tons of stuff for a simple little tutorial.

NEXT

Let's Read In a Picture of a Park

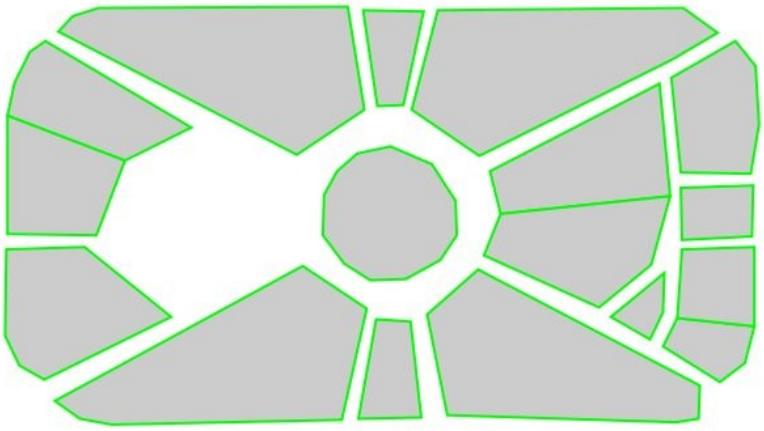
Now that we can create SVG pictures in Haskell, next thing we'll want to do is read them from other programs. Below is a picture of Stanton Park in Washington DC **(grab the file here)** I drew in InkScape.



Here's the code that will load that picture into Haskell:
(again, just stick it to the bottom of the previous code)

```
  let readPoint :: String -> Point
      readPoint s | Just [x,y] <- matchRegex (mkRegex "([0-9.]+),([0-9.]+)") s =
(read x,read y)

  let readPolygon :: String -> Polygon
      readPolygon = (map readPoint).(splitRegex $ mkRegex " L ")

  let readPolygons :: String -> [Polygon]
      readPolygons = (map readPolygon).tail.(splitRegex $ mkRegex "<path")

  park_data <- readFile "park.svg"

  let park = readPolygons park_data

  writeFile "tut1.svg" $ writePolygons (green park)
```

Here's what `tut1.svg` looks like- Notice that only the outlines of the polygons are colored once the park data has made a pass through our program- There's no need in this simple tutorial to track the fill colors separately from the outlines, so we'll just leave those colorless:



This code is pretty much structured the same way as the code that did the writing of the SVG. In this case we're using regular expressions to split the data into parts that have each polygon. The SVG format is actually very complex, so this code takes some liberties in the format and may fail on some SVG files- For the purposes of loading some SVG maps into our program, though, it's great!

What's good about this code?

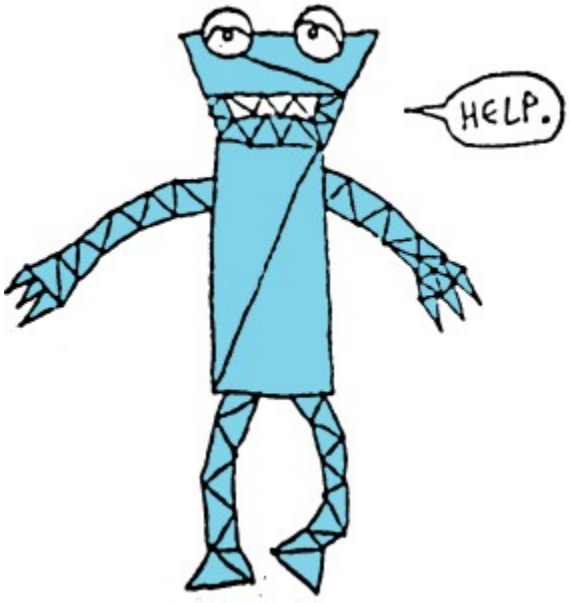
This code illustrates another situation where Haskell's laziness really makes things easy for us: All the regular expression functions just take regular 'ol text strings... If you've ever used regular expression libraries in other languages, you may remember they usually use streams or ports or file handles. Haskell just uses strings. Some of you may protest "Hey! what if you're reading in a 2GB file? Are you just going to read that into a string and then parse it? That would be suicide!"

In most languages, this would indeed be suicide, but not in Haskell: Because it's a lazy language, it won't actually read in any data from the file it doesn't feel it needs- So, theoretically, this code would be just as efficient as reading a regular expression from a stream! Additionally, if you aren't using the `park_data` text string for anything else, the language will probably garbage collect the "front end" of the file as well. So, in theory, we could search through a 2GB file in Haskell in this incredibly simple manner and still maintain a memory footprint that is comparable to that of a stream-based regular expressions library!

NEXT

Alright already!!! Can we finally start to actually organize a picnic???

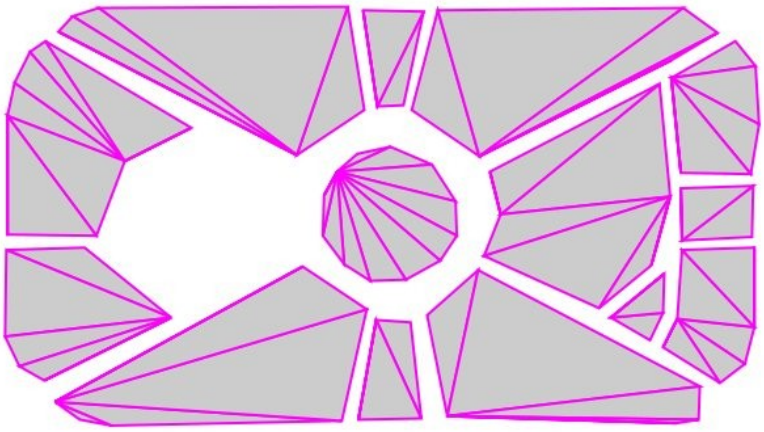
Yes! now that we can read and write SVG pictures, we're finally ready to start organizing a picnic!



First thing we'll want to do is break our city park into evenly-sized little lots for each picnickers to sit on- In order to do this, we'll first need some functions that can slice and dice big polygons into small polygons- The first of these functions is `triangulate`:

```
let triangulate :: Polygon -> [Polygon]
    triangulate (a:b:c:xs) = [a,b,c]:triangulate (a:c:xs)
    triangulate _ = []

let triangles = concatMap triangulate park

writeFile "tut2.svg" $ writePolygons (purple triangles)
```

Here's what the file `tut2.svg` will look like:



There's a few things i'll need to explain about the `triangulate` function...

First of all, you notice that this function is actually has two definitions, with different patterns to the left of the equal sign (`(a:b:c:xs)` and _) The way Haskell works is that it will try to match to the first pattern if it can. Then, if that fails, it'll go to the next one it finds.

In the first version of the function, it checks to see if the list of points has at least three points in it- As you can imagine, it's hard to triangulate something if you don't have at least three points. The colons in the `(a:b:c:xs)` expression let you pick the head item off of a list (or, for that matter, stick something on the head if we used it on the right side of the equation) so this pattern means we need the next three "heads" of the list to be 3 values `a`, `b`, and `c`. If we don't have three points, the second version of the function will match instead. (anything will match the underline character)

If we *do* find that we have three points, the first version of `triangulate` will make a triangle and will then recursively call itself to build more triangles. In a language like Haskell, which has no loops, these types of recursive functions that consumes lists are a classic design. Most of the time, however, we can avoid explicitly creating recursive functions like this by using list functions like `map` and *folds*, which we'll discuss later.

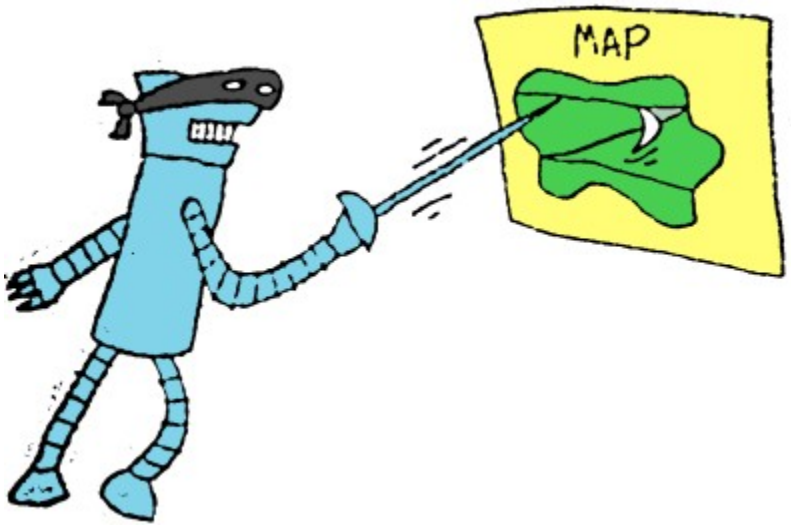
What's good about this code?

Using Haskell pattern matching and recursion, we can very elegantly express functions that process lists, like `triangulate`.

What's bad about this code?

Polygon triangulation is actually slightly more complicated than our function suggests, because there's special procedures that would need to be followed to triangulate concave polygons... In this tutorial, we're skirting this issues by removing convex polygons when we draw our city park maps- That's why there's some oddball extra lines in the original park map.

NEXT



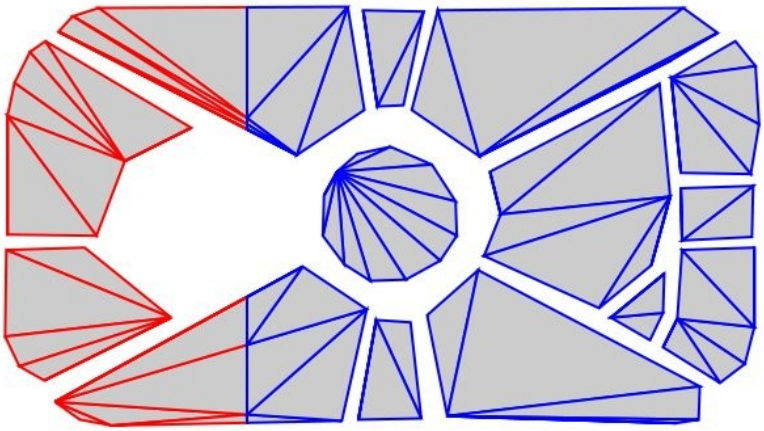
Slicing Up A Park, Neatly

Now that weve broken up our park into little triangles, it's pretty easy to write some functions that will partition the park into chunks by slicing our triangle piles along arbitrary horizontal and vertical lines. Here's the code that will do that:

```
let clipTriangle :: (Point -> Point -> Point) -> [Point] -> [Point] -> [Polygon]
    clipTriangle i [] [a,b,c] = []
    clipTriangle i [a]  [b,c] = [[a,i a b,i a c]]
    clipTriangle i [a,b]  [c] = [[a,i a c,b],[b,i a c,i b c]]
    clipTriangle i [a,b,c] [] = [[a,b,c]]
```

```
  let slice :: (Point -> Bool) -> (Point -> Point -> Point) -> [Polygon] ->
([Polygon],[Polygon])
      slice f i t = (clip f,clip $ not.f)
          where clip g = concatMap ((uncurry $ clipTriangle i).(partition g)) t

  let sliceX :: Float -> [Polygon] -> ([Polygon],[Polygon])
      sliceX x = slice ((x >).fst) interpolateX
          where interpolateX (x1,y1) (x2,y2) = (x,y1+(y2-y1)*(x-x1)/(x2-x1))

  let sliceY :: Float -> [Polygon] -> ([Polygon],[Polygon])
      sliceY y = slice ((y >).snd) interpolateY
          where interpolateY (x1,y1) (x2,y2) = (x1+(x2-x1)*(y-y1)/(y2-y1),y)

  let (left_side,right_side) = sliceX 200 triangles

  writeFile "tut3.svg" $ writePolygons $ (red left_side) ++ (blue right_side)
```

If we look at `tut3.svg`, this is what we'll see:



Let's look at the different functions to see how this works... The `slice` is the heavy lifter in this code snippet: It embodies the abstract action of slicing a pile of polygons using a line. What makes it cool is that it is extremely abstract and general- The actual line is represented by two functions passed into it: One function, of type `(Point -> Bool)`, lets `slice` know if a point is on one side or the of the arbitrary line. The other function, of type `(Point -> Point -> Point)`, lets it know the point at which two points on opposite sides of the line intersect with the line- Our interpolation function.

Think of what this means: By making `slice` a *higher order function* (higher order functions are functions that take other functions as parameters) we can decouple the task of slicing from any details about the location or angle of the cut.

Having the abstract `slice` function makes it easier to write more concrete vertical and horizontal slicing functions (`sliceX` and `sliceY`, respectively.) If we wanted to, we could write functions that slice at other angles nearly as easily, still using `slice` to do the actual cutting.

The `clipTriangle` function is a tool used by `slice`, which figures out if a given triangle is cut by a line and breaks it into 3 baby triangles, if this is the case.

What's good about this code?

We used higher order programming to decouple the general job of slicing triangles along a line from the grimy math involved with specific types of lines. This is another great tool for modularizing our programs that Haskell makes easy.

What's bad about this code?

Actually, I think this code is pretty much *all around good*, given the task at hand.

NEXT

How to Make Sure Everyone Gets a Fair Slice of the Park

Now that we have the tools we need to cut our park into nice little pieces, how do we figure out the best way to cut it up?

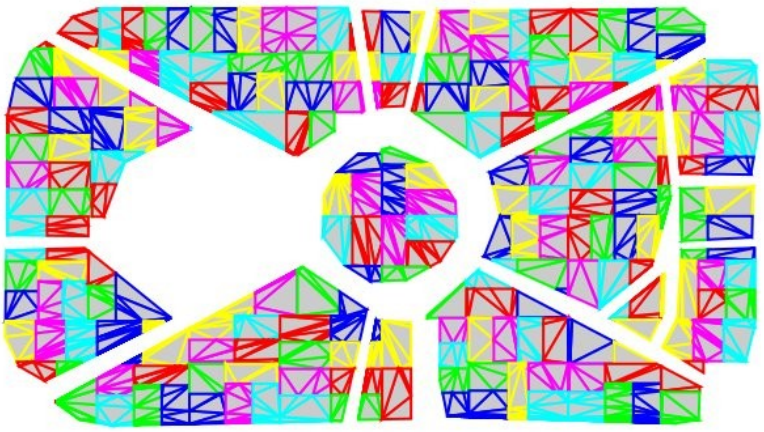
Well, I don't have a perfect answer (if there is any to be had) but it's not too hard to come up with a pretty good solution using a heuristic approach. Here's how we're going to do it:

First, we cut the park roughly down the middle in two halves- If the park is "fat", we cut it vertically... If the park is "tall", we cut it horizontally. Next, we calculate the surface area of each side- Using the ratio of surface areas, we can now randomly break our picnickers into two populations, one for each side.

We keep doing this until every person has their own space- Here's the code that makes it work:

```
let boundingRect :: [Polygon] -> (Float,Float,Float,Float)
    boundingRect p = (minimum xs,minimum ys,maximum xs,maximum ys)
        where xs = map fst $ concat p
              ys = map snd $ concat p

let halveTriangles :: Int -> [Polygon] -> ([Polygon],[Polygon])
    halveTriangles n p = let (l,t,r,b) = boundingRect p
                             f         = fromIntegral n
                             h         = fromIntegral $ div n 2
                         in if r-l > b-t
                            then sliceX ((r*h+l*(f-h))/f) p
                            else sliceY ((b*h+t*(f-h))/f) p

let distance :: Point -> Point -> Float
    distance p1 p2 = sqrt (deltax*deltax+deltay*deltay)
        where deltax = (fst p1)-(fst p2)
              deltay = (snd p1)-(snd p2)

let area :: Polygon -> Float
    area [a,b,c] = let x = distance a b
                       y = distance b c
                       z = distance c a
                       s = (x+y+z)/2
                   in sqrt (s*(s-x)*(s-y)*(s-z))

let allocatePeople :: Int -> [Polygon] -> [[Polygon]]
    allocatePeople 0 t = []
    allocatePeople 1 t = [t]
    allocatePeople n t = let (t1,t2) = halveTriangles n t
                             a1      = sum $ map area t1
                             a2      = sum $ map area t2
                             f = round $ (fromIntegral n)*a1/(a1+a2)
                         in (allocatePeople f t1)++(allocatePeople (n-f) t2)

let lots = allocatePeople (length people) triangles

writeFile "tut4.svg" $ writePolygons $ concat $ zipWith ($) (cycle rainbow) lots
```

Here's what `tut4.svg` looks like- Every picnicgoer get their own picnic lot, of a decent size and shape:



The `halveTriangles` function in this code snippet takes a count of the number of picnic attendees (for the block of land currently being allocated) and cuts the park roughly in half, based on its bounding rectangle. I say "roughly", because we would only want to cut the park exactly in half if the number of people is even- If its odd, then an even cut can be very unfair- Imagine if there were three people left- In that case, an even cut on a rectangular lot would force two of the people to share a single half... It's much smarter if we cut slightly off-center when there's an odd number of people, then it'll be more likely that everyone will get their fair share. The `halveTriangles` function does the math for this- It also compares the length and width of the lot to decide whether a horizontal or vertical cut would lead to "squarer" lots.

The other important function is the allocatePeople function- This is basically the "land commissioner" of this algorithm- It's the function that recursively cuts things into smaller and smaller pieces, and divies up the land chunks to subsets of the people, by comparing land area (via the `area` function, which uses Heron's Law) against the number of people. In the end, everyone gets a pretty fair piece.

What's good about this code?

This kind of code is perfect for a functional programming language, like Haskell- All we're doing is successively churning through all of our park land and handing off to people, something that is easily captured using the recursive

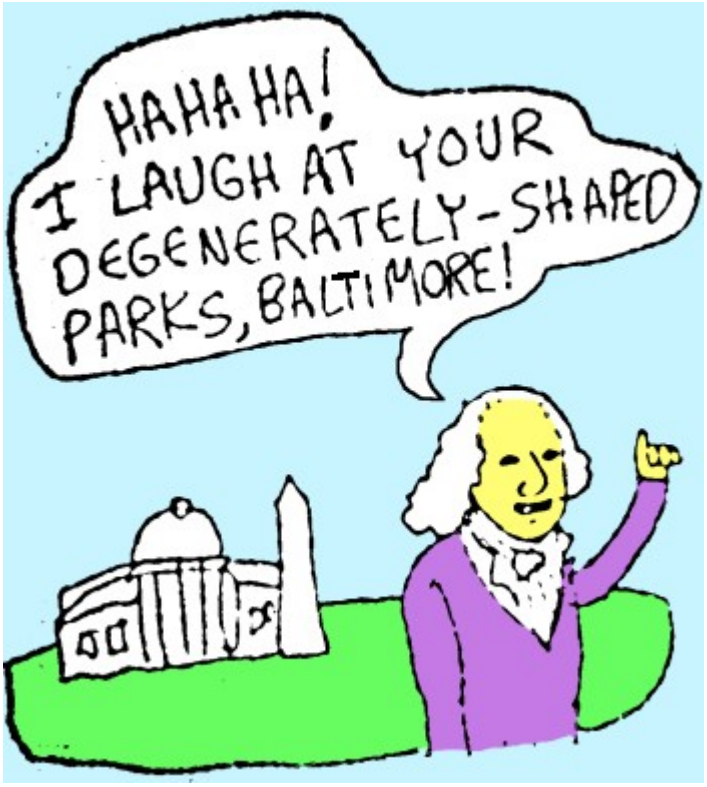
`allocatePeople` function.

Another cool feature of this code is that it colors all the land chunks using a rainbow of colors- We do this by converting our `rainbow` of colors into an infinitely repeating rainbow of colors using the `cycle` function, and then we write a little esoteric code that will call every successive rainbow color function (remember, `rainbow` consisted of a list of color-applying functions) to make a rainbow-colored map of park lots.

What's bad about this code?

First of all, every time we need to check the land area for a region of the park we're recalculating it from scratch- Since this is done over and over again for the same pieces as the park is recursively subdivided, a lot of computrons are being wasted. A better version would "remember" previous triangle areas using memoization.

A second problem with this land subdivision algorithm is that it isn't mathematically perfect... It's just a heuristic: If you look at the final map, you can see small areas that are missing/wasted in the end- You may also be able to come up with certain degenerate park shapes that could break the algorithm altogether. Luckily, here in Washington DC *we don't have no stinkin' degenerately shaped parks*... though I can't vouch for Baltimore. (Just Kidding)



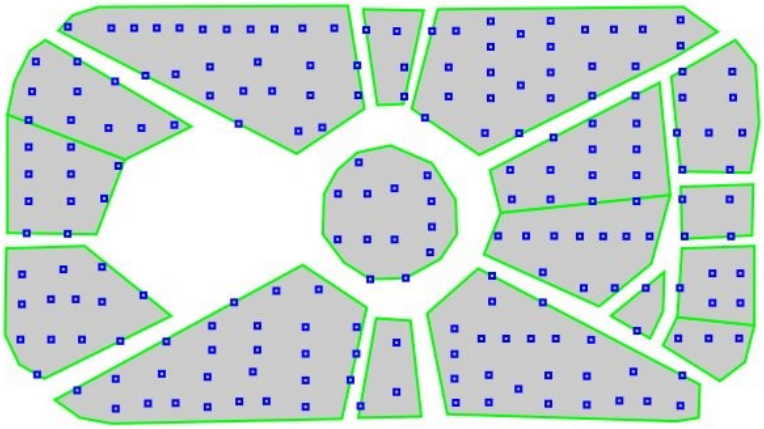
NEXT

Making Sure No One Sits in The Middle of a Walkway

Once we have our equal-area lots determined, we could ideally just stick each picnic blanket right in the middle of the lot. However, even though most lots will be square-ish in shape, given that the original shapes can have walkways and stuff, we'll want to make sure that a lot with a walkway in it doesn't have the center in a walkway. Here's the code that finds a *smarter* center and draws a dot in it:

```
  let findLotCenter :: [Polygon] -> Point
      findLotCenter p = let (l,t,r,b) = boundingRect p
                            m@(x,y)   = ((r+l)/2,(b+t)/2)
                            (lh,rh)   = sliceX x p
                            (th,bh)   = sliceY y $ lh ++ rh
                            centerOrder p1 p2 = compare (distance p1 m) (distance
p2 m)
                        in minimumBy (comparing $ distance m) $ concat $ th ++ bh

  let makeDot :: Point -> Polygon
      makeDot (x,y) = [(x-2,y-2),(x+2,y-2),(x+2,y+2),(x-2,y+2)]

  let centers = map findLotCenter lots

  let spots = blue $ map makeDot centers

  writeFile "tut5.svg" $ writePolygons $ (green park) ++ spots
```

Here's what `tut5.svg` looks like:



The trick in the `findLotCenter` is to use our old `sliceX` and `sliceY` functions to make one more "plus-sign" cut and then pick the central-most vertex from the resulting triangles.

NEXT

Calculating the Neighbors For Every Picnic Spot

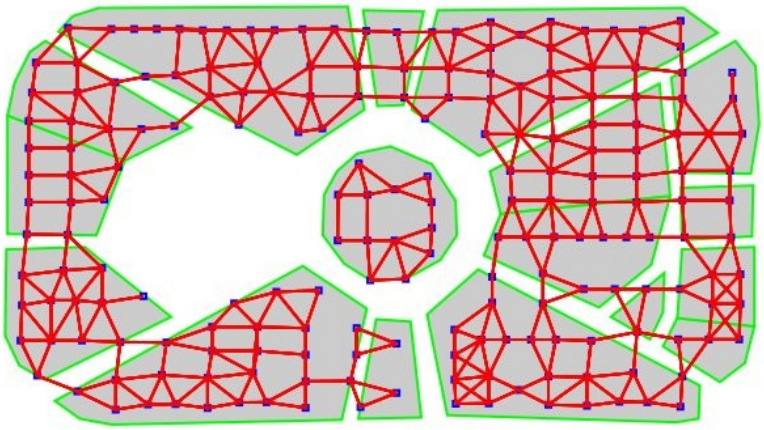
We're going to be finding an optimal placement of our picnickers using *simulated annealing*. This means, all the picnickers are going to "run around" like crazy on the picnic map and when they find people they like, they're going to move slower and "crystalize"... It will be as if the picnickers are molecules in a beaker that are slowly forming a crystal.

In the process of simulating this, we'll need to know what the neighboring spots are for every picnic spot on the map- We'll need to know for two reasons: First, once we start putting people in the spots, we'll need a way to tell how well they'll like their neighbors to make them slow down to crystalize. Second, while all the picnickers are randomly "bouncing around" in the park, we need to know what spots the can "bounce into" next. Here is some code that calculates neighbors:

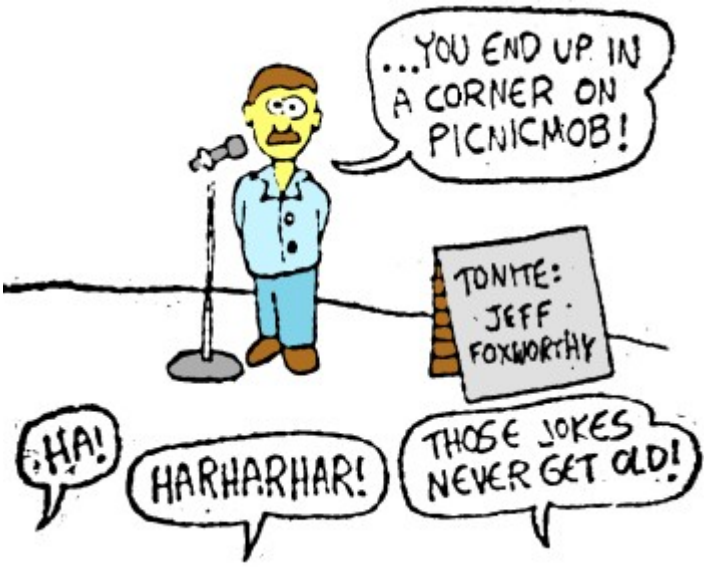
```
let shortestLinks :: Int -> [Link] -> [Link]
    shortestLinks n = (take n).(sortBy $ comparing linkLength)
     where linkLength [a,b] = distance a b

let sittingNeighbors :: Int -> [Point] -> [Link]
```

```
      sittingNeighbors n p = nub $ shortestLinks (n * (length p)) [[a,b] | a <- p,
b <- p, a /= b]

  let sitting = sittingNeighbors 4 centers

  writeFile "tut6.svg" $ writePolygons $ (green park) ++ spots ++ (red sitting)
```

Here's what the resulting `tut6.svg` file will look like:



The way neighbors are determined is to simply check every possible picnic spot pair, sort them, and select the shortest. Naturally, this means that centrally-located spots will have more neighbors (and would probably fetch top dollar, if it wasn't for this $#&@! real estate market right now...) while the edge spots will have less- Which actually nicely handles the cases of outliers in the picnicmob questionaire... *If you end up in a corner of the park on picnicmob.org you're probably some kind of weirdo.* ...Uhm... Wait... I mean... it's probably just coincidence, really- forget I said that...



What's good about this code?

If you look at the `sittingNeighbors` function (which builds all possible *spot pairs* and sorts 'em) you see something in square brackets `[ ]`... This is a neat feature in Haskell called a *list comprehension*... In this example, it reads as "For the cartesian products of all spots, represented as spots *a* and *b*, where those spots are not identical, return a the list of a-b pairs". Since lists are so central to Haskell, being able to write stuff that generates lists in this fashion allows some algorithms to be expressed very clearly and succinctly.

What's Bad About This Code?

Somewhere, deep in the brain of every programmer, is a neuron that has only one role in life: Whenever a sound is recorded by the hair cells of the ear that sounds like "Cartesian Product", this neuron will dump unimaginable quantities of neurotransmitters at the nodes of other neurons, that translate into the neuron-equivalent of **"Oh my God!! *INEFFICIENCY ALARM RED!!* All Neurons to full alert! RUN RUN RUN!!!"** Enough said.

Another problem, looking at the resulting map, is that this `sittingNeighbors` function is prone to lead to "islands"... this is fine if when we're measuring happiness with neighbors when people are sitting (since people on other sides of walkways aren't really that good for conversation,

anyway...) but is lousy for letting our molecule-people bounce around- They'll never walk/bounce into the islands... Which is why we need to define another function for neighbors... you can probably guess what it'll be called...

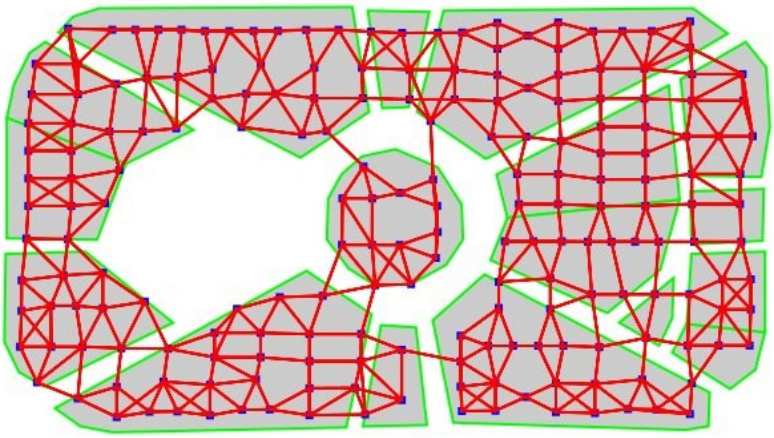
NEXT

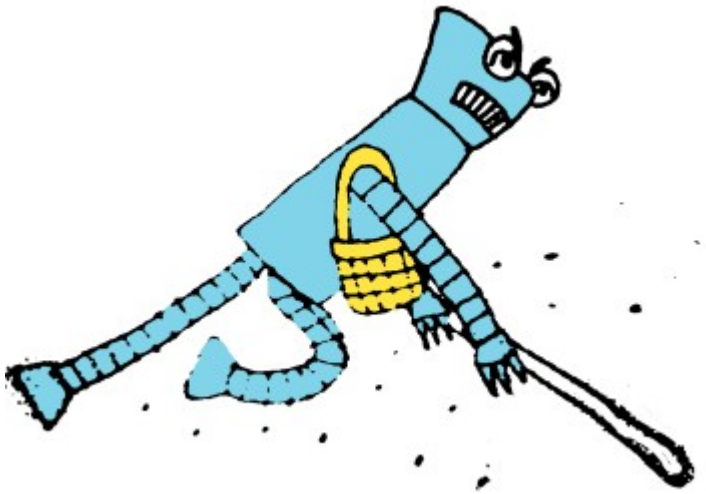
Finding "Walkable" Neighbors

This function is just a variant of the last one that just make sure at least four of a spot's nearby nodes are represented in the list of neighbor links:

```
let walkingNeighbors :: Int -> [Point] -> [Link]
    walkingNeighbors n l = nub $ concatMap myNeighbors l
        where myNeighbors :: Point -> [Link]
              myNeighbors p = shortestLinks n [sort [p,c] | c <- l, p /= c]

let walking = walkingNeighbors 4 centers

writeFile "tut7.svg" $ writePolygons $ (green park) ++ spots ++ (red walking)
```

Note that with this function, the islands are now more like peninsulas (or perhaps *jetties*... or maybe *enclaves* would be more appropriate... definitely not *buttes*, though...) This is the function we'll use for "walking around".



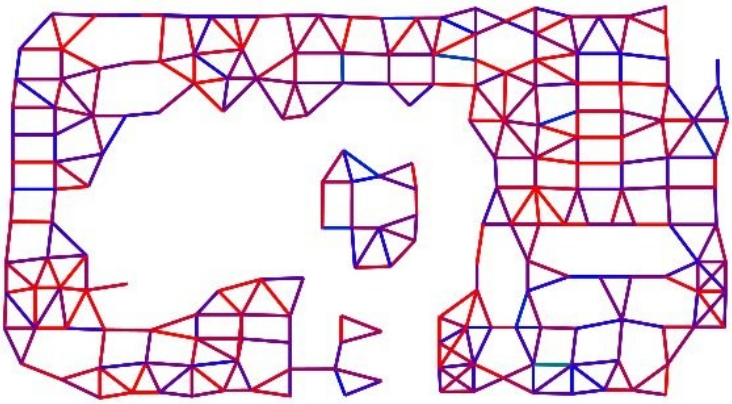


On your mark...

Let's place people in their starting position now and draw a pretty map showing people's happiness at the start... Here's the code for this:

```
  let starting_placement = zip centers people

  let mismatches :: Person -> Person -> Int
      mismatches a b = length $ filter (uncurry (/=)) $ zip a b

  let similarityColor :: Person -> Person -> Color
      similarityColor p1 p2 = let m = mismatches p1 p2
                                  h = div (length p1) 2
                                  d = 30 * (abs (h - m))
                                  b = max 0 (255-d)
                                  o = min d 255
                              in if m < h
                                 then (0,o,b)
                                 else (o,0,b)

  let findPerson :: Placement -> Point -> Person
      findPerson a p | Just (_,e) <- find ((== p).fst) a = e

  let similarityLine :: Placement -> Link -> (Color,Polygon)
      similarityLine l [p1,p2] = (similarityColor (findPerson l p1) (findPerson l
p2),[p1,p2])

  writeFile "tut8.svg" $ writePolygons $ map (similarityLine starting_placement)
sitting
```

Here's what `tut8.svg` looks like:



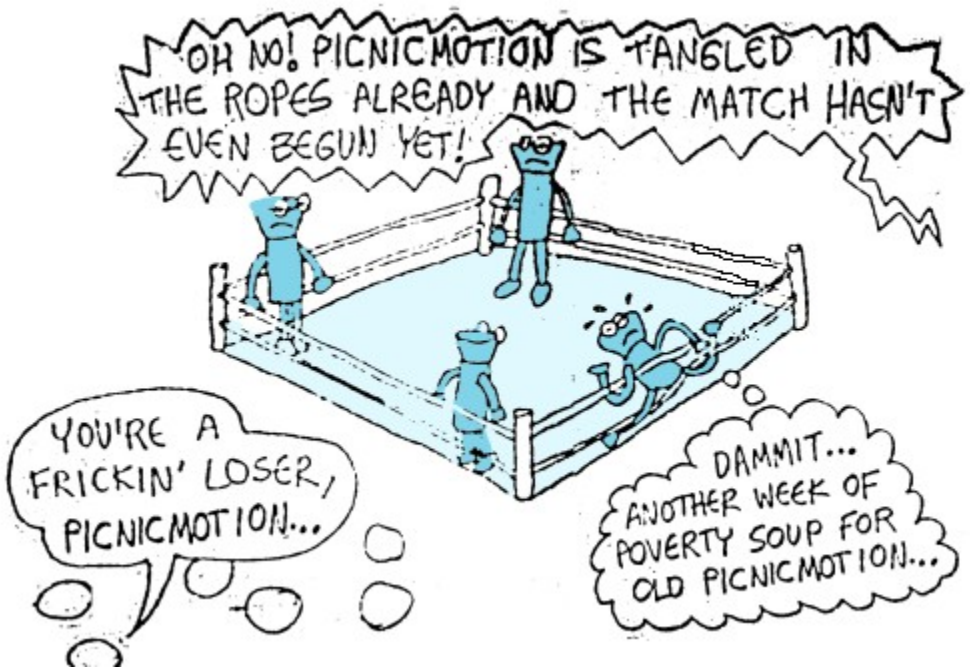
The only really important function in this block of code is the first line that calculates the `starting_placement`... This is just a `zip`ped together list of the lot centers and people into spot-people pairs. The rest of the code is just for the eye candy...

The color of the lines tells you how well two neighbors get along... The `similarityColor` function is a little kludge that creates a nice color from red to blue, depending on how compatible two people are (as usual in Haskell, the type signature of `Person->Person->Color` is pretty much a dead give-away...)

NEXT

Get Set...

Next, we need to define our four simulated annealing functions... In the left corner, `picnicEnergy` tells us the overall happiness of our picnic population... smaller, low energy values mean greater happiness. In the right corner, `picnicMotion` swaps two neighbors randomly to cause motion. In the front corner, (This is a tag-team match) `picnicTemperature` gives the current temperature: We want to start hot, then cool things down so that our picnic crystalizes in a controlled manner. And finally, in the back, picnicTransitionProbability takes the current temperature and the energy of two states and calculates the likelihood a transition into the new state will occur. For more info, consult [you know what](#)...




```
  let picnicEnergy :: [Link] -> EnergyFunction Placement
      picnicEnergy l a = sum $ map linkEnergy l
          where linkEnergy :: Link -> Int
                linkEnergy [p1,p2] = mismatches (findPerson a p1) (findPerson a
p2)

  let picnicMotion :: [Link] -> MotionFunction Placement
      picnicMotion l r a = let (n,r2) = randomR (0,(length l)-1) r
                               [p1,p2] = l!!n
                           in (r2,(p1,findPerson a p2):(p2,findPerson a p1):
(filter (not.((flip elem) [p1,p2]).fst) a))

  let picnicTemperature :: TemperatureFunction
      picnicTemperature m c = 50.0 * (exp (0.0 - (5.0 * ((fromIntegral c) /
(fromIntegral m)))))
```

```
let picnicTransitionalProbability :: TransitionProbabilityFunction
    picnicTransitionalProbability e1 e2 t = exp ((fromIntegral (e1 - e2)) / t)

let annealing_time = 500

putStr "starting energy: "
print $ picnicEnergy sitting starting_placement

putStr "starting temperature: "
print $ picnicTemperature annealing_time annealing_time
```

When we run the program with this new addition, We'll now get some info on the picnic starting positions:

```
> runHaskell tutorial.hs

Hello World! Let's have a picnic!
Number of people coming:
200
starting energy: 16010
starting temperature: 0.33689734
```

What's Good About This Code?

When it came to the four simulated annealing functions, it was very easy to take the text from the wikipedia, generically translate the types of these functions into generic haskell types, and then create instances of these functions concretely defined for our picnic-organizing needs. This really shows the power of Haskell's highly refined support for declaring and manipulating types.

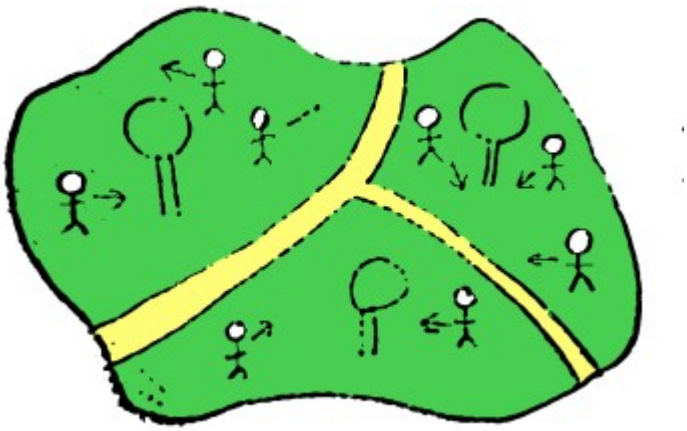
What's Bad About this Code?

Well, these four functions are going to be the bread and butter of our this program- Probably, about 99% of CPU time will be spent in this little bit of code... Consequently, every little inefficiency in this code will hurt performance hundredfold... And boy, is this code inefficient: Because we're storing everything in lists, it means that any function doing a search, such as `findPerson`, will be inefficient to the point of absurdity... Even worse is the fact that picnicEnergy checks the happiness of the *entire picnicker population*, even though

only two people move at any one time and does so in the most tedious way by checking every questionaire question on every person against neighbors over and over and over again, instead of precalculating the compatibility of two people ahead of time.

These things can be fixed relatively easily by ugli-fying the code with extra parameters holding precalculated/memoized info and using GHC `Maps`, `HashMaps`, and `Arrays` instead of POLs (plain old lists, to possible coin a new term- 'pwned', watch your back... a new kid's in town!) in the appropriate places. In fact, the "real" version of this annealer I'm using for my picnicmob calculations does all this, but ain't exactly prime tutorial material. However, it runs well over a thousand times faster than this simplified version :-)

NEXT



GO!!!

Alright! now we're finally anealling us picnic! here's our main "loop":

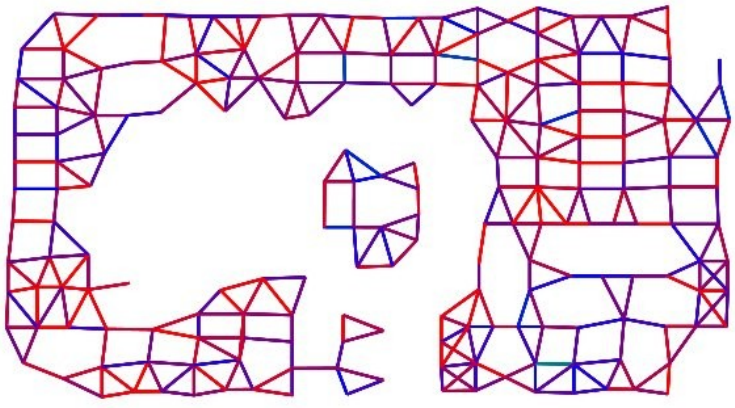
```
  let anneal_tick :: MotionFunction a -> TransitionProbabilityFunction ->
EnergyFunction a -> Float -> (StdGen,a) -> (StdGen,a)
      anneal_tick mf tpf ef t (r,p) = let (r2,p2) = mf r p
                                          (n ,r3) = random r2
                                      in (r3,
                                          if n < tpf (ef p) (ef p2) t
                                          then p2
                                          else p)

  let anneal :: EnergyFunction a -> MotionFunction a ->
```

```
TransitionProbabilityFunction -> TemperatureFunction -> Int -> StdGen -> a -> a
      anneal ef mf tpf tf m r s = snd $ foldl' (flip (anneal_tick mf tpf ef))
(r,s) (map (tf m) [0..m])

  random_generator <- getStdGen

  putStr "starting annealing... "
  putStr "number of annealing steps: "
  print annealing_time

  let ideal_placement = anneal
                        (picnicEnergy sitting)
                        (picnicMotion walking)
                        picnicTransitionalProbability
                        picnicTemperature
                        annealing_time
                        random_generator
                        starting_placement

  writeFile "tut9.svg" $ writePolygons $ map (similarityLine ideal_placement)
sitting

  putStr "Done!\nfinal energy: "
  print $ picnicEnergy sitting ideal_placement
  putStr "final temperature: "
  print $ picnicTemperature 0 annealing_time
```

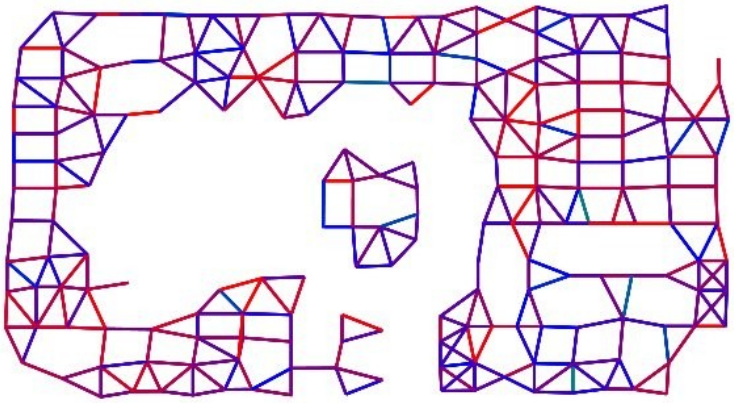
Now, if we run our the completed program, we'll calculate ourselves low energy, crystallized picnic exactly as we wanted: *(be patient it'll might take 5 minutes for a result...)*

```
> runHaskell tutorial.hs

Hello World! Let's have a picnic!
Number of people coming: 200
starting energy: 16010
starting temperature: 0.33689734
starting annealing... number of annealing steps: 500
Done!
final energy: 15010
final temperature: 0.0
```

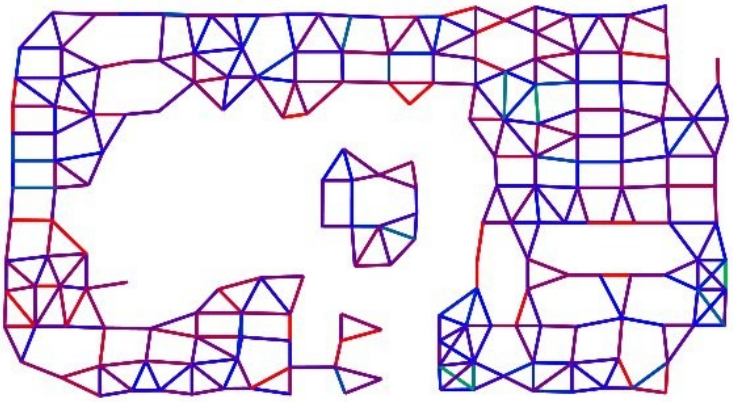
Before



After



Here's what it looks like with a much longer annealing time:



WooHoo! Now let's figure out what this final piece of code is all about...

First, we have the `anneal_tick` function, which handles a single moment in time for our annealing... It needs to be handed three of the four annealing functions... Instead of the

fourth one, `TemperatureFunction`, it is handed just the temperature at that moment (the `Float`), since at any given time the temperature is just a single number. The last thing passed into this function is the current placement of our "atoms", as well as a random number source, `StdGen`... In Haskell, you can't just pull random numbers out of "thin air" as you can in almost any other programming language known to mankind... Remember, *doing unpredictable stuff that isn't specified explicitly in a function's type signature is a Haskell no-no*...

The main "loop" of our program is in the function `anneal`... I put "loop" in quotes because Haskellers don't use loops, they use *folds*... A fold is kind of like a `map` function: Whereas map returns a list created by thrashing through a starting list and applying a function to each member, folds return a single item, created by *folding together* all the items in a list into a single result value- There are two main folding functions, `foldl` and `foldr`, which fold from the left and right end of the starting list, respectively. To learn more about the different types of folds, check the HaskellWiki.

Finally, we code generates the `ideal_placement` by glueing together all our building block functions to generate our final result- And that's the end of our tutorial program!

Of course, the total number of annealing steps we're doing (500) is not enough for a very good annealing- You'd need to run a few million steps and use GHC to compile the program to machine language to get an optimal result- Here's how you'd compile it:

```
ghc -O2 -fglasgow-exts -optc-march=pentium4 -optc-O2 -optc-mfpmath=sse -optc-msse2
--make picnic.hs
```

What's Good About this Code?

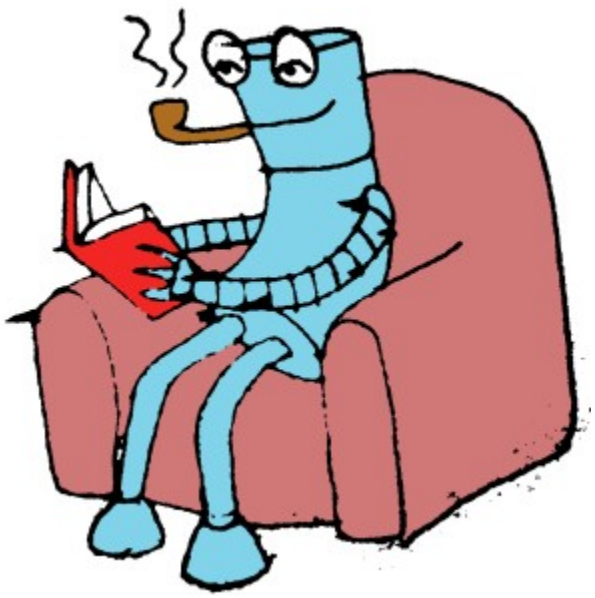
Once again, this shows how haskell's typing system let's us modularize like a charm: Note that the word "picnic" or anything else picnic-related does not appear in either the `anneal_tick` or the `anneal` function... This code, which is the very heart of this program, is completely unpolluted by our problem domain: If you were a biochemist simulating artery clot annealing, you could use this exact same code to run your simulations. If you're a nuclear chemist simulating isotope doodads in a hydrogen bomb, you could use this function. *(Note to nuclear chemists- If you're reading this*

tutorial, and copy/pasting my code, then we're in big trouble...)

As this example shows, Haskell's powerful typing system allows us to prevent leakage from different sections of code in ways almost no other language can match- Whether you're worried about leaking randomness, IO operations, picnic references, etc. etc. Haskell's type system offers the ultimate in "code leakage" protection!

What's Bad About this Code?

In conclusion of my tutorial, I want to wander off a bit into the realm of armchair programming philosophy...



I, like many others involved in Haskell, believe very strongly that the central ideas in the Haskell language represent the future of programming- It's seems pretty inevitable that Haskell-like will have a huge impact on the future... But, alas, Haskell still has one unavoidable weakness that might be the fly in the ointment of the inevitable Haskell utopia of the future- We can see it clearly in the code above where this weakness comes in to play...

Several times in this tutorial I have talked about how easy it is to take algorithms or math concepts and effortlessly translate them into elegant types and functions using the elegance of the Haskell syntax and typing system. One set of functions, however, were not so easy to translate: the

`anneal_tick` and `anneal` functions. The source of these functions was the *pseudo-code* in the wikipedia article we've been using... They're just a wee-bit more obfuscated than one would really want, in my opinion. It's only a little bit obfuscated, but still makes me unhappy...

The reason for this is that this pseudo code is simulating a physical world that changes slightly over time: This is a fundamental property of our universe: **Things only change a little bit from one moment to the next.** A great description of this noteworthy property of our world can be found in this Whitepaper which discusses a pretty cool take on A.I. that is being explored by Jeff Hawkins and Dileep George at a company named Numenta- Their software is all about how our minds exploit this property in achieving intelligence...

Because the physical world changes only slightly from moment to moment, it means that languages that can comfortably mutate large data structures in targeted ways will always have a role to play in real-world software- The "real world" usually just doesn't work the way Haskell, and other functional languages would prefer it did: Haskell preferred that at every moment in time, a "new universe" would *look at* the "old universe" and would rebuild itself, from scratch, from what it saw in the past, with radical changes happening all the time.

Despite its many advantages, I humbly suggest, therefore, that in the future there will continue to be a rift between the "imperative" and "functional" camps of programming, until someone comes up with a truly robust way of uniting these two camps- And I think that some profound programming discoveries still need to be made in the future before this problem is really resolved- I get the feeling it's just not good enough to wave at the problem and say **"Monads"**.

Back To lisperati.com