Guards in Functions

A function to find the "sign" of a number:

```
sgn x = if x>0 then 1 else if x<0 then -1 else 0
```

Here is a slick way to write it, using guards:

```
sgn x | x>0 = 1
      | x<0 = -1
      | otherwise = 0
```

Bindings in `where` clauses are visible in guards:

```
f x | w > 10 = 1
    | otherwise = 2
  where w = x*x
```

Polymorphism

Recall the types of lists:

```
[True, False] :: [Boolean]
[Rectangle 1 2, Ellipse 1 2] :: [Shape]
[] :: [what should go here?]
```

Whatever type [] has, it must be consistent with these:

```
True : [] :: [Boolean]
Rectangle 1 2 : [] :: [Shape]
```

The first expression requires [] to have type [Boolean].

The second expression requires [] to have type [Shape].

How could this be possible?

Polymorphism

The type of `[]` is `[a]`. The `a` here is called a *type variable*.

Note that a type variable begins in lower case. (An actual type begins in upper case.)

A type variable can stand for any type. It is instantiated to an actual type so as to satisfy the context. E.g.,

```
True : []              --a is instantiated to Boolean
Rectangle 1 2 : []     --a is instantiated to Shape
```

If the context does not impose any type on `a`, it remains uninstantiated. E.g.,

```
[]  --has type [a] when alone
```

In this way, `[]` is *polymorphic*: it can have any of a multitude of types.

Polymorphic Function

A function to count the elements in a list.

```
length [] = 0
length (x:xs) = 1 + length xs
```

Its type is `[a] -> Integer` because nothing in the function determines the type of the list elements.

This is a *polymorphic function*: its parameters (and even return values) can have any of a multitude of types. E.g.,

```
length [True, False]  --parameter is [Boolean]
length [3, 4]         --parameter is [Integer]
length []             --parameter is [a]
```

You can see that polymorphism is Haskell's way of providing *genericity*.

Map

Let's say we have a squaring function:

```
square n = n*n
```

and we want to use it to square every element of a list, e.g., if we have a list `[1,3,5]`, we want to get `[1,9,25]`. We might write:

```
squareList [] = []
squareList (x:xs) = square x : squareList xs
```

Now let's say we have a `cube` function and we want to do the same:

```
cubeList [] = []
cubeList (x:xs) = cube x : cubeList xs
```

This gets boring after a few more examples. Isn't there a better way?

Map

The Haskell library has a `map` function. If you want to apply a function `f` to every element of a list `xs`, you do this:

```
map f xs
```

Here is how `map` looks like; note how it generalizes `squareList` and `cubeList`:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Let us consider the type of map. An element `x` may be of type `a`, and `f` may map it to type `b`. Thus `f :: a->b`, the input list is `[a]`, and the output list is `[b]`. Then

```
map :: (a -> b) -> [a] -> [b]
```

Higher-Order Functions

The function

```
map :: (a -> b) -> [a] -> [b]
```

takes a parameter that is in turn a function.

In general, functional languages allow a function to take functions as parameters and even return functions as return values. Such a function is called a *higher-order function*.

One more example: takes a function `f` and returns a slightly modified function g that does $g(x) = f(x) + 1$.

```
upOne :: (Int -> Int) -> (Int -> Int)
upOne f = g
  where g x = f x + 1
```

(blank page)