Guards in Functions

A function to find the “sign” of a number:

\[ \text{sgn } x = \begin{cases} 
1 & \text{if } x > 0 \\
-1 & \text{if } x < 0 \\
0 & \text{otherwise}
\end{cases} \]

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

\[ \text{sgn } x \mid x > 0 = 1 \\
\mid x < 0 = -1 \\
\mid \text{otherwise } = 0 \]

Bindings in \textit{where} clauses are visible in guards:

\[ f \ x \mid w > 10 = 1 \\
\mid \text{otherwise } = 2 \\
\text{where } w = x^2 \]
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.

\[
\text{length \ [\ ]} = 0 \\
\text{length \ (x:xs)} = 1 + \text{length \ xs}
\]

Its type is \([a] \rightarrow \text{Integer}\) 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.,

\[
\begin{align*}
\text{length \ [True, False]} & \quad \text{--parameter is \ [Boolean]} \\
\text{length \ [3, 4]} & \quad \text{--parameter is \ [Integer]} \\
\text{length \ [\]} & \quad \text{--parameter is \ [a]}
\end{align*}
\]

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

Let’s say we have a squaring function:

\[ \text{square } n = n \times n \]

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:

\[
\text{squareList } [] = [] \\
\text{squareList } (x:xs) = \text{square } x : \text{squareList } xs
\]

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

\[
\text{cubeList } [] = [] \\
\text{cubeList } (x:xs) = \text{cube } x : \text{cubeList } xs
\]

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

The Haskell library has a \texttt{map} function. If you want to apply a function $f$ to every element of a list $xs$, you do this:

\[
\text{map } f \text{ } xs
\]

Here is how \texttt{map} looks like; note how it generalizes \texttt{squareList} and \texttt{cubeList}:

\[
\text{map } f \text{ } [] = [] \\
\text{map } f \text{ } (x:xs) = f \text{ } x : \text{map } f \text{ } xs
\]

Let us consider the type of \texttt{map}. An element $x$ may be of type $a$, and $f$ may map it to type $b$. Thus $f :: a \rightarrow b$, the input list is $[a]$, and the output list is $[b]$. Then

\[
\text{map :: (a } \rightarrow \text{ b) } \rightarrow \text{ [a] } \rightarrow \text{ [b]}
\]
Higher-Order Functions

The function

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

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 \( g(x) = f(x) + 1 \).

\[
\text{upOne} :: (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})
\]
\[
\text{upOne} \ f = g
\]
\[
\text{where} \ g \ x = f \ x + 1
\]
(blank page)