Higher Order Functions

Haskell allows the arguments of functions to be other functions. We can also have a function return another function as its value. Functions that manipulate other functions are called *higher order functions* and we will see that this can be a very useful mechanism.

As an introduction to higher order functions, we will examine some examples of manipulating words and sentences. Consider the example of getting the first letter of every word in a sentence:

```
firstLetters :: Language -> Language
firstLetters s = if (empty s)
                 then s
                 else (firstItem (firstItem s)) +++ (firstLetters (butFirst s))

> firstLetters (sent "american civil liberties union")
[a c l u]
```

As another example, suppose we wish to square all the number that are contained in a sentence

```
squareSent :: Language -> Language
squareSent s = if (empty s)
               then s
               else (squareWord (firstItem s)) +++ (squareSent (butFirst s)) where
               squareWord w = if (wordIsNum w)
                              then intToWord (square (wordToInt w))
                              else w

> squareSent (sent "1 2 3 and 4")
[1 4 9 and 16]
```

You should see a strong similarity in the solutions of these two examples. In both, we applied a function (`firstItem` or `squareWord`) to every word in the sentence. This is a very common pattern so we will apply an *abstraction* to capture this recurring pattern. We can write a higher order function which does this, where the function applied to every word in the sentence (of type `Language -> Language`) is passed in as a parameter. A slightly fancier version of this function `every` is supplied in the Words module:

```
every :: (Language -> Language) -> Language -> Language
every f s = if (empty s)
            then s
            else (f (firstItem s)) +++ (every f (butFirst s))

> every firstItem (sent ("american civil liberties union")
[a c l u]
```

```
> every squareWord (sent "1 2 3 and 4")
[1 4 9 and 16]
```

Another common task that can be encapsulated into a higher order function is to select only the words in a sentence that satisfy a given predicate. The predicate is a function that returns true or false for a given word `(Language -> Bool)` and is the first parameter of the function `keep`. A slightly fancier version of this function `keep` is supplied in the Words module:

```
keep :: (Language -> Bool) -> Language -> Language
keep test s = if (empty s)
            then s
            else if test (firstItem s)
            then (firstItem s) +++ (keep test (butFirst s))
            else keep test (butFirst s)

> keep wordIsNum (sent ("4 calling birds 3 french hens 2 turtle doves")
[4 3 2]

> keep isVowel (word "piggies") where isVowel letter = member letter (word "aeiou")
iie
```

Yet another common task is to accumulate the words in a sentence using some sort of combiner function. The combiner is a function that takes two words and combines them into one so it signature is `(Language -> Language -> Language)` and is the first parameter of the function `accumulate`. A slightly fancier version of this function `keep` is supplied in the Words module:

```
accumulate :: (Language -> Language -> Language) -> Language -> Language
accumulate combine s = if (count s) == 1
            then (firstItem s)
            else combine (firstItem s) (accumulate combine (butFirst s))

> accumulate (+++)  (sent ("a c l u")
aclu

> accumulate addNum (sent "1 2 3")  where
          addNum x y =  intToWord (wordToInt x + wordToInt y)
6
```