# Functional Programming - Passing a function as a parameter (Haskell)

One of the core principles of Functional Programming (FP) is that functions are ‘first-class objects’ - meaning, amongst other things, that a function can be passed as a parameter into another function, or returned as the result of another function.

In this exercise we are going explore what passing a function as a parameter looks like in code and, more importantly, the benefits that that possibility offers.

Let us start by imagining that you have written a function that sorts a list of strings into alphabetical order. You might have implemented another of the standard algorithms, such as Insertion or Quick Sort - or perhaps you’ve even designed a brand new high-performance algorithm. The example code below implements the Merge Sort algorithm. It is actually two functions: SortAlphabetical, which splits a list, and then delegates to a second function to merge them. Both functions are recursive:

mergeAlphabetical a b =

if a == []

then b

else if b == []

then a

else if head a < head b

then head a : mergeAlphabetical (tail a) b

else head b : mergeAlphabetical a (tail b)

sortAlphabetical list =

if length list < 2

then list

else mergeAlphabetical (sortAlphabetical (drop (div (length list) 2) list))

(sortAlphabetical (take (div (length list) 2) list))

We can prove that this works either by calling the SortAlphabetical function from within a Console program, or a unit test, for example:

testSortAlphabeticalHappyCase =

assertEqual ["Burg","Cup","Flag","Nest", "Next","Yacht"]

(sortAlphabetical ["Flag","Nest","Cup","Burg", "Yacht","Next"])

Now, suppose that we want to change the program to sort the names in reverse alphabetical order, this would require a change to just one line of code, where we compare two string values, from:

else if head a < head b

to

else if head a > head b

But what if, as is more likely, we want the option to sort *either* way?

This is the cue for ‘passing a function as a parameter’. In the code below we’ve renamed the two functions to just Sort and Merge respectively - partly because this reflects their more generic capability and partly to allow us to keep the old and new versions alongside each other in the same file without a clash:

merge a b greaterThan =

if a == []

then b

else if b == []

then a

else if greaterThan a b

then head a : merge (tail a) b greaterThan

else head b : merge a (tail b) greaterThan

sort list greaterThan =

if length list < 2

then list

else merge (sort (drop (div (length list) 2) list ) greaterThan ) (sort (take (div (length list) 2) list) greaterThan) greaterThan

As in our previous version, both functions take an additional parameter to specify *how* we want the list sorted, but this time it is not a simple Boolean, but rather a function, called greaterThan. The type of this parameter is defined as:

greaterThan :: string -> string -> Boolean

Which may be read as ‘a function that takes in two strings as parameters and returns a Boolean result’. Each of the following, separate, standalone, functions fits this specification:

alphabetical a b = a < b

reverseOrder a b = a > b

longer a b = (length a) > (length b)

Notice that each of these three functions has a different name, but they all have the same type signature to fit the requirements of the greaterThan function needed as the second parameter for the new sort function.

So we can now test the sort function using any of those three, or any other function that has the same type signature:

testSortWithAlphabeticalFunction =  
 assertEqual ["Burg","Cup","Flag","Nest", "Next","Yacht"]  
 (sort ["Flag","Nest","Cup","Burg", "Yacht","Next"] alphabetical)

testSortWithReverseFunction =  
 assertEqual ["Yacht", "Next", "Nest", "Flag", "Cup","Burg"]  
 (sort ["Flag","Nest","Cup","Burg", "Yacht","Next"] reverseOrder)

This Haskell implementation can not just sort strings, but any type of object. The following code shows the same function now being used to sort a list of integers, first in increasing, then in decreasing order:

testSortIntegers =

assertEqual [2,3,4,7,7,9,12,88]

(sort [4,7,12,3,88,9,2,7] alphabetical)

testSortIntegersInReverse =

assertEqual [88,12,9,7,7,4,3,2]

(sort [4,7,12,3,88,9,2,7] reverseOrder)

Note that ‘alphabetical’ is therefore not a good name, and should be re-named to, say, ‘ascending’.