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

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:

let rec mergeAlphabetical (a: List<string>) (b: List<string>) =

if a.IsEmpty then

b

else if b.IsEmpty then

a

else if a.Head < b.Head then

a.Head :: mergeAlphabetical a.Tail b

else

b.Head :: mergeAlphabetical a b.Tail

let rec sortAlphabetical (list: List<string>) =

if list |> Seq.length < 2 then

list

else

let half = (list |> Seq.length) / 2

mergeAlphabetical (sortAlphabetical (list |> List.skip(half)))

(sortAlphabetical (list |> List.take(half)))

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

[<TestMethod>]

member this.TestSortAlphabeticalHappyCase() =

let list = ["Flag";"Nest";"Cup";"Burg"; "Yacht";"Next"]

let sorted = sortAlphabetical list

let expected = ["Burg";"Cup";"Flag";"Nest"; "Next";"Yacht"]

Assert.AreEqual(expected, sorted)

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 a.Head < b.Head then

to

else if a.Head > b.Head then

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:

let rec merge (a: List<string>) (b: List<string>) greaterThan =

if a.IsEmpty then

b

else if b.IsEmpty then

a

else if greaterThan a.Head b.Head then

b.Head :: (merge a b.Tail greaterThan)

else

a.Head :: merge a.Tail b greaterThan

let rec sort (list: List<string>) greaterThan =

if list |> Seq.length < 2 then

list

else

let half = (list |> Seq.length) / 2

merge (sort (list |> List.skip(half)) greaterThan)   
 (sort (list |> List.take(half)) 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:

member this.alphabetical s1 s2 = s1 > s2

member this.reverse s1 s2 = s1 < s2

member this.length (s1:string) (s2: string) = s1.Length > s2.Length

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:

[<TestMethod>]

member this.TestSortWithAlphabeticalFunction() =

let list = ["Flag";"Nest";"Cup";"Burg"; "Yacht";"Next"]

let sorted = sort list this.alphabetical

let expected = ["Burg";"Cup";"Flag";"Nest"; "Next";"Yacht"]

Assert.AreEqual(expected, sorted);

[<TestMethod>]

member this.TestSortWithReverseFunction() =

let list = ["Flag";"Nest";"Cup";"Burg"; "Yacht";"Next"]

let sorted = sort list this.reverse

let expected = ["Yacht"; "Next"; "Nest"; "Flag"; "Cup";"Burg"]

Assert.AreEqual(expected, sorted)

[<TestMethod>]

member this.TestSortByLengthDecreasing() =

let list = ["Flag";"Yachting";"Cup";"Burger"; ]

let sorted = sort list this.length

let expected = ["Cup"; "Flag"; "Burger";"Yachting"]

Assert.AreEqual(expected, sorted);

We don’t even need to code the implementation of ‘greaterThan’ as a separate standalone function: we can define it a ‘lambda’ - which is just like a function declared in-line. The following example of using a lambda, produces the same result as using the pre-defined length function (above):

[<TestMethod>]

member this.TestSortByLengthDecreasingUsingLambda() =

let list = ["Flag";"Yachting";"Cup";"Burger"; ]

let sorted = sort list (fun (s1:string) (s2: string) -> s1.Length > s2.Length)

let expected = ["Cup"; "Flag"; "Burger";"Yachting"]

Assert.AreEqual(expected, sorted);

In the above code the lambda (s1, s2) => s2.Length > s1.Length may be read as ‘*Given* two strings, s1 and s2, *return* the result of s2.Length > s1.Length. The ‘->’ symbol is sometimes pronounced as ‘goes to’.

Typically, you would only define an explicit standalone function for performing the string comparison, if you wanted to be able to use that same comparison more than once. (Using Lambda’s has some other advantages, too, but these are outside the scope of this lesson).

It is also now possible for us to generalise our mergesort function further, so that it can not just sort strings, but any type of object. To do this we use the ‘generics’ syntax, where we specify the type of object being sorted as ‘T’, and where we must now provide a ‘greaterThan’ function that takes in two objects of type T and returns a Boolean:

let rec merge (a: List<'T>) (b: List<'T>) greaterThan =

if a.IsEmpty then

b

else if b.IsEmpty then

a

else if greaterThan a.Head b.Head then

b.Head :: (merge a b.Tail greaterThan)

else

a.Head :: merge a.Tail b greaterThan

let rec sort (list: List<'T>) greaterThan =

if list |> Seq.length < 2 then

list

else

let half = (list |> Seq.length) / 2

merge (sort (list |> List.skip(half)) greaterThan)   
 (sort (list |> List.take(half)) greaterThan) greaterThan

The following code shows the same function now being used to sort a list of integers, first in increasing, then in decreasing order:

[<TestMethod>]

member this.TestSortIntegers() =

let list = [4; 7; 12; 3; 88; 9; 2; 7]

let sorted = sort list this.greaterThan

let expected = [2;3;4;7;7;9;12;88]

Assert.AreEqual(expected, sorted);

[<TestMethod>]

member this.TestSortIntegersInReverse() =

let list = [4; 7; 12; 3; 88; 9; 2; 7]

let sorted = sort list this.reverse

let expected = [88;12;9;7;7;4;3;2]

Assert.AreEqual(expected, sorted);

Making use of these two implementations of the greaterThan function *specifically* for comparing two integers:

member this.greaterThan s1 s2 = s1 > s2

member this.reverse s1 s2 = s1 < s2

(In fact the reverse function is the same one used for sorting strings in reverse order - it just works in F#).

The MergeSort example that we’ve worked through here is a simple, but nonetheless realistic case for passing a function as a parameter. In fact, there are standard libraries that do exactly this. For example, the following code calls the standard Microsoft Seq (sequence) function sortBy, passing in a lambda to determine what to order by:

let list = ["Flag";"Nest";"Cup";"Burg"; "Yacht";"Next"]

let sorted = list |> Seq.sortBy(fun s -> s.Length)