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

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. (FList is a custom implementation of a ‘functional list’ - meaning that the list is immutable, and is made up of a Head and Tail – but that is not our concern here). It is actually two functions: SortAlphabetical, which splits a list, and then delegates to a second function to merge them. Both functions are recursive:

Public Shared Function SortAlphabetical(list As FList(Of String)) As FList(Of String)

If list.Count() < 2 Then

Return list

Else

Dim half = list.Count() / 2

Return MergeAlphabetical(SortAlphabetical(list.Skip(half)),

SortAlphabetical(list.Take(half)))

End If

End Function

Public Shared Function MergeAlphabetical(a As FList(Of String), b As FList(Of String))

As FList(Of String)

If a.IsEmpty Then

Return b

ElseIf b.IsEmpty Then

Return a

ElseIf String.Compare(a.Head, b.Head) < 0 Then

Return FList.Cons(a.Head, MergeAlphabetical(a.Tail, b))

Else

Return FList.Cons(b.Head, MergeAlphabetical(a, b.Tail))

End If

End Function

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

<TestMethod> \_

Public Sub TestSortAlphabeticalHappyCase()

Dim list = FList.Cons("Flag", "Nest", "Cup", "Burg", "Yacht", "Next")

Dim sorted = MergeSort.SortAlphabetical(list)

Dim expected = FList.Cons("Burg", "Cup", "Flag", "Nest", "Next", "Yacht")

Assert.AreEqual(expected, sorted)

End Sub

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:

ElseIf String.Compare(a.Head, b.Head) < 0 Then

to

ElseIf String.Compare(a.Head, b.Head) > 0 Then

But what if, as is more likely, we want the option to sort *either* way? The simplest way to do this, in terms of programmer effort, would be to duplicate the original two methods, rename them to, say, SortReverse and MergeReverse, and change the line doing the comparison as shown above.

However, duplicating and then slightly modifying code is *always* a bad idea – the algorithm is essentially the same but now we have twice as much code to test and maintain. If, in future, we find a way to improve the algorithm we need to remember to alter it in both places.

A better option would be to pass in an additional Boolean parameter indicating whether we want the sorting to be alphabetical or reverse. For example:

Public Shared Function SortAlphabetical(list As FList(Of String), reverse As Boolean)

As FList(Of String)

If list.Count() < 2 Then

Return list

Else

Dim half = list.Count() / 2

Return MergeAlphabetical(SortAlphabetical(list.Skip(half), reverse),

SortAlphabetical(list.Take(half), reverse), reverse)

End If

End Function

Public Shared Function MergeAlphabetical(a As FList(Of String), b As FList(Of String),

reverse As Boolean) As FList(Of String)

If a.IsEmpty Then

Return b

ElseIf b.IsEmpty Then

Return a

ElseIf (Not reverse And String.Compare(a.Head, b.Head) < 0) Or

(reverse And String.Compare(a.Head, b.Head) > 0) Then

Return FList.Cons(a.Head, MergeAlphabetical(a.Tail, b, reverse))

Else

Return FList.Cons(b.Head, MergeAlphabetical(a, b.Tail, reverse))

End If

End Function

This is definitely a better solution: we’ve avoided duplicating all the common code. But now suppose another requirement comes along for a list of words to be sorted by word-length. We could expand the previous pattern and accommodate the new requirement, but the code will start to get ugly, and the risks increase that we will accidentally break one of the other forms of sorting in the process (though we could certainly mitigate that risk by having comprehensive automated tests). What we want is a single version of our sort function where we can somehow just change a small part of the code inside the function - in this case just the one line where we compare two strings. 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:

Public Shared Function Sort(list As FList(Of String),

greaterThan As Func(Of String, String, Boolean)) As FList(Of String)

If list.Count() < 2 Then

Return list

Else

Dim half = list.Count() / 2

Return Merge(Sort(list.Skip(half), greaterThan),

Sort(list.Take(half), greaterThan), greaterThan)

End If

End Function

Public Shared Function Merge(a As FList(Of String), b As FList(Of String),

greaterThan As Func(Of String, String, Boolean)) As FList(Of String)

If a.IsEmpty Then

Return b

ElseIf b.IsEmpty Then

Return a

ElseIf greaterThan(a.Head, b.Head) Then

Return FList.Cons(a.Head, Merge(a.Tail, b, greaterThan))

Else

Return FList.Cons(b.Head, Merge(a, b.Tail, greaterThan))

End If

End Function

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 As Func(Of 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:

Private Shared Function alphabetical(s1 As String, s2 As String) As Boolean

Return String.Compare(s2, s1) > 0

End Function

Private Shared Function reverse(s1 As String, s2 As String) As Boolean

Return String.Compare(s2, s1) < 0

End Function

Private Shared Function length(s1 As String, s2 As String) As Boolean

Return s2.Length > s1.Length

End Function

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> \_

Public Sub TestSortWithAlphabeticalFunction()

Dim list = FList.Cons("Flag", "Nest", "Cup", "Burg", "Yacht", "Next")

Dim sorted = MergeSort.Sort(list, AddressOf alphabetical)

Dim expected = FList.Cons("Burg", "Cup", "Flag", "Nest", "Next", "Yacht")

Assert.AreEqual(expected, sorted)

End Sub

<TestMethod> \_

Public Sub TestSortWithReverseFunction()

Dim list = FList.Cons("Flag", "Nest", "Cup", "Burg", "Yacht", "Next")

Dim sorted = MergeSort.Sort(list, AddressOf reverse)

Dim expected = FList.Cons("Yacht", "Next", "Nest", "Flag", "Cup", "Burg")

Assert.AreEqual(expected, sorted)

End Sub

<TestMethod> \_

Public Sub TestSortByLengthDecreasing()

Dim list = FList.Cons("Flag", "Nest", "Cup", "Burg", "Yacht", "Next")

Dim sorted = MergeSort.Sort(list, AddressOf length)

Dim expected = FList.Cons("Cup", "Flag", "Nest", "Burg", "Next", "Yacht")

Assert.AreEqual(expected, sorted)

End Sub

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>

Public Sub TestSortByLengthDecreasingUsingLambda()

Dim list = FList.Cons("Flag", "Nest", "Cup", "Burg", " Yacht ", "Next")

Dim sorted = MergeSort.Sort(list, Function(s1, s2) s2.Length > s1.Length)

Dim expected = FList.Cons("Cup", "Flag", "Nest", "Burg", "Next", " Yacht ")

Assert.AreEqual(expected, sorted)

End Sub

In the above code the lambda Function(s1, s2) s2.Length > s1.Length) may be read as ‘*Given* two strings, s1 and s2, *return* the result of s2.Length > s1.Length.

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:

Public Shared Function Sort(Of T)(list As FList(Of T),

greaterThan As Func(Of T, T, Boolean)) As FList(Of T)

If list.Count() < 2 Then

Return list

Else

Dim half = list.Count() / 2

Return Merge(Sort(list.Skip(half), greaterThan),

Sort(list.Take(half), greaterThan), greaterThan)

End If

End Function

Public Shared Function Merge(Of T)(a As FList(Of T), b As FList(Of T),

greaterThan As Func(Of T, T, Boolean)) As FList(Of T)

If a.IsEmpty Then

Return b

ElseIf b.IsEmpty Then

Return a

ElseIf greaterThan(a.Head, b.Head) Then

Return FList.Cons(a.Head, Merge(a.Tail, b, greaterThan))

Else

Return FList.Cons(b.Head, Merge(a, b.Tail, greaterThan))

End If

End Function

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

<TestMethod> \_

Public Sub TestSortIntegers()

Dim list = FList.Cons(4, 7, 12, 3, 88, 9, 2, 7)

Dim sorted = MergeSort.Sort(list, AddressOf greaterThan)

Dim expected = FList.Cons(2, 3, 4, 7, 7, 9, 12, 88)

Assert.AreEqual(expected, sorted)

End Sub

<TestMethod> \_

Public Sub TestSortIntegersInReverse()

Dim list = FList.Cons(4, 7, 12, 3, 88, 9, 2, 7)

Dim sorted = MergeSort.Sort(list, AddressOf reverse)

Dim expected = FList.Cons(88, 12, 9, 7, 7, 4, 3, 2)

Assert.AreEqual(expected, sorted)

End Sub

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

Private Shared Function greaterThan(i1 As Integer, i2 As Integer) As Boolean

Return i2 > i1

End Function

Private Shared Function reverse(i1 As Integer, i2 As Integer) As Boolean

Return i1 > i2

End Function

The 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, which uses a regular list rather than our functional list, calls the standard Microsoft Linq function OrderBy, passing in a lambda to determine what to order by:

Dim list = New List(Of String)() From

{"Flag", "Nest", "Cup", "Burg", "Yacht", "Next"};

Dim sorted = list.OrderBy(Function(s) s.Length)