Practical 3

CSc302 Functional Programming

Deadlines:

- **Practical** section: **7 March** at **5pm**
- **Homework** section: **12 March** at **2pm**

In this practical, we will focus on the use of function techniques.

Warm-up

1. Create a new F# file in Visual Studio and save it, with the extension **.fsx**, in a location that you know, so that you can get back to it later. You will do all your work in this file.

List-related higher-order functions

One of the common uses of higher-order functions is to specialise the operations that are commonly performed with data structures. We will introduce some of the most common of those operations in this section. It will be **extremely important** to understand the *meaning* of these operations so that you can recognize them—in fact, it will be so important that I have created a set of emphasised blocks that define them in this document and that you should pay attention to.

map

The **map** operation independently transforms all elements in a structure, returning a transformed structure.

Key words / phrases:

- "**independently**": each element is transformed by itself, without looking at any other item.
- "**element**": an individual data item within a structure
- "**all**": the transformation is applied to every element, without exception.
- "**transformed structure**": a structure which is exactly the same as the input structure, *except* for the fact that a transformation has been applied to each element independently.

	Inputs: 1. The function which transforms an element. 2. The structure that contains elements. Output: ● The transformed structure.

1. A **map** function is one of the most useful ones that you will create, and one of the simplest. It independently transforms all elements in a structure, returning a transformed structure. We can implement it for lists in this way:

```
1  let rec listMap f =
2    function
3    | [] -> []
4    | h::t -> f h :: listMap f t
```

2. Push it down to F# Interactive. You should see that its type is **('a -> 'b) -> List<'a> -> List<'b>** , so you can see that it has an arity of 2. Play with it a bit. Perhaps try these little experiments:
   a. listMap (fun a -> a + 2) [0; 2; 3; 4]
   b. listMap (fun b -> (b, b)) ["yo"; "lo"; "hi"; "ho"]
   c. listMap (fun c -> if c % 2 = 0 then "even!" else "odd!") [9; 0; 2; 1; 4]
3. Note that the <u>**order of elements stays the same**</u>. The map does not change the *structure* of the list at all: it leaves the ordering alone. It just transforms the elements of the list.
4. The "unusual" line of **listMap** is line 4, where we see the code **f h** . **f** is a symbol that bound to a function. **h** is an element, as can be seen from the pattern on line 4. Therefore, all that this "unusual" code does is apply a function to an element. As we know, when we apply a function to an element, it must give us its output. So, this generates an output, and the **:: listMap f t**—which is a recursive call that passes the function and the tail to **listMap**—will eventually result in a list to which it will be "cons"ed.
5. Of course, the "map" function is a general *pattern of computation* that you will see over, and over, and over again. You can write a map for *any* data structure, as long as you can sensibly identify a structure that contains elements and have some way of traversing through those elements. We created a Tree<'a> structure previously:

```
type Tree<'a>
  = EmptyTree
  | TwoChildren of 'a * Tree<'a> * Tree<'a>
  | LeftOnly of 'a * Tree<'a>
  | RightOnly of 'a * Tree<'a>
  | NoChildren of 'a
```

6. What would a **treeMap** look like? Perhaps something like this:

```
let rec treeMap f =
  function
  | EmptyTree →
    EmptyTree
  | TwoChildren (v, l, r) →
    TwoChildren (f v, treeMap f l, treeMap f r)
  | LeftOnly (v, l) →
    LeftOnly (f v, treeMap f l)
  | RightOnly (v, r) →
    RightOnly (f v, treeMap f r)
  | NoChildren v →
    NoChildren (f v)
```

7. Note that after mapping, the tree might not be suitable as a binary search tree: the map operation works with *individual values* and does not alter the *structure of the tree*. Assuming that a tree named **myTree** exists, can you see which of these operations will cause a tree to remain as a binary search tree, and which will not?
    a. treeMap (fun x -> x + 1) myTree
    b. treeMap (fun y -> y % 10) myTree

That concludes our introduction to the "map" operation. We now consider the "filter" operation:

filter

The **filter** operation omits all independent elements which fail a predicate function, returning a structure without such elements.

Key words / phrases:

- "**omits**": some elements are left out of the final structure.
- "**independent**": each element is considered by itself, without looking at any other item.
- "**predicate function**": a function which returns a boolean value.
- "**fail a predicate function**": when the predicate function is applied to this element, it returns false
- "**all**": the filtering is applied to every element, without exception.

Inputs:

1. A predicate function that accepts an element.
2. The structure that contains elements.

Output:

- The filtered structure.

8. What could a filter look like, for lists? Perhaps it might look something like this:

```
1  let rec listFilter p =
2    function
3    | [] → []
4    | h::t →
5        if p h then
6          h :: listFilter p t
7        else
8          listFilter p t
```

9. Once again, we see the use of a function (line 5). This time, applying the **p** function to the element will supply the **if**-expression with the necessary boolean value that it needs to decide which branch to take.
10. Play with the function a bit! Try these out:
    a. listFilter (fun x -> x % 2 = 0) [9;0;2;1;0]
    b. listFilter (fun y -> y % 2 = 1) [9;0;2;1;0]
    c. listFilter (fun z -> z < 22) [9;0;2;1;0]
11. Notice that once again, we do not change the *structure* of the list. All elements which are not omitted are kept in the same relative order that they had in the original list.
12. Can we write a filter for a Tree<'a>? It is tempting to think that we can do this easily. However, if we try, we will have to change the structure of the tree, and that is problematic since a **filter** operation is supposed to leave the structure intact. For example, let us assume that one has a **TwoChildren(v, l, r)** value, and the value **v** fails the predicate. Obviously, the node must be omitted—but what should we do about the **l** and **r** cases? They might *not* be omitted, so they have to be included. However, no node can have an empty parent, so some sort of structural change will inevitably be necessary.

    If we are willing to accept the smallest structural change that we can get away with, then we may be able to do this, using some functions that we've already created. One of those will be **treeNLR**, and another will be **treeInsert**, which should look something like this (if your **treeInsert** *wasn't* working, please study this code and your code *carefully* so that you can understand why this approach worked and your approach didn't!):

```
let rec treeInsert v =
 function
 | EmptyTree → NoChildren v
 | NoChildren x →
```

```
  if v < x then
    LeftOnly (x, NoChildren v)
  elif v > x then
    RightOnly (x, NoChildren v)
  else
    NoChildren x
| LeftOnly (x, xs) ->
  if v < x then
    LeftOnly (x, treeInsert v xs)
  elif v > x then
    TwoChildren (x, xs, treeInsert v EmptyTree)
  else
    LeftOnly (x, xs)
| RightOnly (x, xs) ->
  if v < x then
    TwoChildren (x, treeInsert v EmptyTree, xs)
  elif v > x then
    RightOnly (x, treeInsert v xs)
  else
    RightOnly (x, xs)
| TwoChildren (x, xs, ys) ->
  if v < x then
    TwoChildren (x, treeInsert v xs, ys)
  elif v > x then
    TwoChildren (x, xs, treeInsert v ys)
  else
    TwoChildren (x, xs, ys)
```

13. Our idea is as follows: if we have a tree **t0**, and we traverse it in NLR[1] order to obtain a list, and then create a tree **t1** based on the list, then **t0** = **t1**. Why? Because insertion of nodes in the same order must result in the same tree; and because traversal in NLR order gives you a plausible order in which nodes were inserted.

    Think carefully about this, draw some trees, and convince yourself that it is

[1] Or NRL order.

true—try and find a case where it isn't true, if you'd like!—and when you've seen that it must be true, continue reading.

14. To obtain the minimum number of structural changes, let us get the NLR-order list from the tree, filter the list, and then recreate the tree using only the nodes which aren't omitted. We already have a function to get the NLR-order list; and we have a function to filter that list; but we *don't* yet have a function that will accept a list and generate a tree. Let's create one:

```
let listToTree list =
 let rec makeTree tree =
   function
   | [] → tree
   | h::t → makeTree (treeInsert h tree) t
 makeTree EmptyTree list
```

15. Now we have all the necessary functions in place. Let's make a **treeFilter**. Before reading on, think about all the functions we've created, and see if you can figure out what **treeFilter** will look like. It's surprisingly simple!

    To see if you're right, have a look at my **treeFilter** below:

```
let treeFilter p tree =
  listToTree (listFilter p (treeNLR tree))
```

    Did your **treeFilter** look similar to this?

The last "common" operation that we will deal with is the "fold" operation.

fold

The **fold** operation combines the elements of a data structure, traversing it in a defined order to build up a new data structure.

Key words / phrases:

- "**combines**": uses a function that accepts a new data structure and an element from the old data structure, and returns a new data structure that appropriately combines the element.
- "**new data structure**": a data structure[2] built up by the combining function.
- "**traversing**": going through the data structure, one element at a time.
- "**defined order**": the order in which elements are

[2] This can also be a "primitive"/ "basic" data structure, such as an integer.

	traversed is fixed.
	Inputs: 1. A combining function that accepts a new data structure and an element. 2. The initial value of the new data structure. 3. The structure that contains elements to be traversed. Output: ● The new data structure.

A "combining function" seems like a weird sort of idea, but let's try a few simple cases and see what we can do with the idea.

16. This funny-looking code is what a "left fold" for a list looks like; study it and examine the description above, and see if you can figure out how it works and what it does:

```
1  let rec listFoldl f acc =
2    function
3    | [] -> acc
4    | h::t -> listFoldl f (f acc h) t
```

17. We call this a **left** fold because it traverses elements in left-to-right order as it goes through the list. That is why there is a "l" at the end of the word "fold". But how does it work?

    The first thing that you might note is that *unlike* in the case of a "map" or "filter" for lists, we aren't appending results to a list on line 4. So, what are we doing?
    a. The code **f acc h** applies the function **f** with the inputs **acc** (the new data structure) and **h** (the current element of the list). The result of this is a new data structure. In other words, **f** is a function that will combine an element with our new data structure, generating another new data structure as a result.
    b. Also on line 4, we have a recursive call to **listFoldl** which passes through the combining function **f**, the result of the latest combination (i.e. **f acc h**), and the rest of the list.
    c. Eventually, we will reach the end of the list. When this happens, we will return the last "new data structure" value that was generated.
18. A **right** fold—which traverses a list right-to-left—looks surprisingly similar, with one line changed:

```
1  let rec listFoldr f acc =
2    function
3    | [] -> acc
4    | h::t -> f (listFoldr f acc t) h
```

19. This looks a bit weird; what is going on??

    Let's unpack it.

    On line 4, the first thing that has to be calculated is **listFoldr f acc t**. That calls the **listFoldr** function recursively, and each time, we pass through the same **f** value and the same **acc** value with a smaller list (i.e. **t**). At some point, eventually, we will get to the end of the list (line 3), at which point we will return **acc**—i.e., the *original value* of **acc** that was passed in.

    And then our stack starts to unwind, and we apply **f** (on line 4) to this original **acc** value, and to the *last* element of the list **h**. (remember that we only reach line 3 when we have reached the end of the list, and therefore the **f** will only be applied *after* the end of the list is reached). So, **f** combines **acc** with **h** to generate a data structure that is the result of the function—and which is passed back as the result to another call made from line 4, this time combining it with the second-last element **h**.

    The process continues until eventually, one reaches the very first (i.e. the leftmost) value of the list, and then the result of that is the result of the entire function.
20. All this talk of "combining functions" is a bit funny. Let's put it to use.

    (***Note***: unlike with "map" and "filter", don't go looking for places to use "fold". At this point in your education, you are unlikely to be able to intuitively identify such places. Instead, after you've created a recursive function, spend a few seconds looking at it and think: could you have done it with a "fold"? If you think the answer might be "yes", then try it and see!)

    Can you see what these examples do? Try them out and see. (the answers are in footnotes)
    a. listFoldl (fun n _ -> n+1) 0 [5; 6; 9; 2; 1; 0; 3] [3]
    b. (function
       | [ ] -> 0
       | h::t -> listFoldl (fun c x -> if x > c then x else c) h t
) [8; 2; 6; -7; 99; 5] [4]
    c. listFoldl (fun s x -> $"{s}{x}") "" ["hi"; "ho"; "yo"; "lo"] [5]
    d. listFoldr (fun s x -> $"{s}{x}") "" ["hi"; "ho"; "yo"; "lo"] [6]

[3] This returns the length of the list.
[4] Finds the maximum value in the list. I should, arguably, be using **None** and **Some** here…
[5] Concatenates all the strings, going from left to right
[6] Concatenates all the strings, going from right to left

21. Just as previously, let's ask the question: can we do this for trees? The answer is a resounding "yes"—BUT of course, trees don't have a natural "left-to-right" or "right-to-left" traversal order. However, we can make a **treeFoldNLR** or a **treeFoldLNR** or a **treeFoldRNL**, since those are natural and defined orders for trees. Let's make just one of them:

```
let rec treeFoldNLR f acc =
  function
  | EmptyTree → acc
  | NoChildren x → f acc x
  | LeftOnly (x, xs) → treeFoldNLR f (f acc x) xs
  | RightOnly (x, xs) → treeFoldNLR f (f acc x) xs
  | TwoChildren (x, xs, ys) →
    treeFoldNLR f (treeFoldNLR f (f acc x) xs) ys
```

22. If we wish to, we could then create a **treeNLR** function with much less code:

```
let treeNLR t = treeFoldNLR (fun acc x → acc @ [x]) [] t
```

23. Folds can typically be used to create very short functions that do a lot. For example, perhaps you remember how much code it took to reverse a list. Would you like to do it in 1 line instead?

```
let reverse l = listFoldl (fun s x → x::s) [] l
```

Or how about our **listToTree** function? We could rewrite it as

```
let listToTree xs =
  listFoldl (fun s x → treeInsert x s) EmptyTree xs
```

That concludes the warm-up. Hopefully, you can see the use of some of these common higher-order functions, you have an idea of how they are made, and are ready to make some of your own.

Practical [30]

In this section:

- you may use functions that you have already defined (including those from the warm-up), but no built-in functions;
- you may create as many functions as you want, to achieve your goals.
- **answers that do not use or develop functions that accept functions will have their mark capped at 3.**

1. [5] Create a **treeFoldLNR** function.

2. [5] Create a **treeExists** function that returns true if a value that meets the supplied predicate exists in the tree.
3. [5] Create a **listForAll** function which accepts a predicate and returns true only if *all* the items in the list pass the predicate. An empty list returns **false**.
4. [5] Create a **listPartition** function that returns a 2-tuple of lists containing values which pass the predicate, and values which do not.
5. [5] Create a **listMapIndexed** function that acts like a map, but also passes the index of the element to the specialist function.
6. [5] Create a **listCollect** function that applies a list-generating specialist function, and joins all the resulting lists together.

Upload your .fsx file to RUConnected.

Homework [17]

1. [5] Create a **treeExcept** function which accepts two trees (**t0** and **t1**) and returns a tree that contains all values that exist in **t0** but not **t1**.
2. [5] Create an **isBST** function that returns true if the tree in question is a binary search tree (i.e. if it obeys all the rules of a binary search tree).
3. [5] A shop sells chocolates for ***r*** rands each, and will exchange ***e*** chocolate wrappers for a new chocolate. Assuming that you start off with **c** rands, how many chocolates will you be able to obtain from the shop? Create a recursive function to solve this problem.
    a. When **c**=15, ***r***=1, ***e***=3, you should get 22
    b. When **c**=16, ***r***=2, ***e***=2, you should get 15
    c. When **c**=12, ***r***=4, ***e***=4, you should get 3
    d. When **c**=6, ***r***=2, ***e***=2, you should get 5
4. [2] Convert your solution from (3) into a fold-using function instead of an explicitly recursive function.

Upload your .fsx file to RUConnected.