Practical 2

CSc302 Functional Programming

Deadlines:

- **Practical** section: **29 February** at **5pm**
- **Homework** section: **5 March** at **2pm**

In this practical, we will focus on the use of recursion, patterns, and recursive functions.

Warm-up

1. Create a new F# file (NOT A NOTEBOOK!) in Visual Studio. You can do this by using the menus and selecting File → New File… → Text File, then clicking on *Select a language* and choosing F#.
2. Save your file, 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.
   a. The ".fsx" extension says that this is a F# script file.
3. Install the **Ionide for F#** extension, if you haven't done so already. This should give you some reasonable Intellisense and allow you to use the "Send to F# Interactive" command (which you can invoke by selecting code and pressing Alt+Enter).
   a. Note that Ionide takes a few seconds—on slow machines, close to a minute—to start up after VS Code appears on-screen. Be patient. After it's started up, you should be able to use the Alt+Enter trick. In the labs, start-up shouldn't take more than 5-10 seconds.

Recursive practice

Before we get into data structures, let's practise our recursion. A prime number is one that is divisible only by itself and by 1. So if we want to know whether a number ***n*** is prime, we can work our way through the numbers [2, ***n***), and if ***n*** divides into any of them cleanly, then it can't possibly be a prime number.

4. This is an easy recursive function. In your file, type:

```
let rec checkPrime number x =
  if number ⩽ 1 || number % x = 0 then
    false
  elif x ⩾ number then
    true
  else
    checkPrime number (x + 1)
```

Note that as usual, I'm using a font that *displays* an operator like ">=" as "≥". You will type ">=". (If you also want to use this font, you can find the instructions here.)

Double-check yourself to ensure that you can see how the recursion is working.

5. If you want to check whether the function works, here are some tests that you can copy-paste into your file.

```
( checkPrime 17 2 = true
, checkPrime 25 2 = false
, checkPrime 99 2 = false
, checkPrime 97 2 = true
, checkPrime 2 2 = true
, checkPrime 1 2 = false
, checkPrime 0 2 = false
, checkPrime -5 2 = false
, checkPrime -39 2 = false
)
```

6. One problem with this way of doing things is that it's ugly. We have to keep passing our "start" number, 2, to the function. Let's make a version of the function that doesn't have this problem:

```
let betterCheckPrime number =
  let rec checkPrimeCandidate x =
    if x ≥ number then
        true
    elif number % x = 0 then
        false
    else
        checkPrimeCandidate (x + 1)
  if number ≤ 1 then
    false
  else
    checkPrimeCandidate 2
```

Compile this so that you can satisfy yourself that it compiles cleanly. There are several aspects of this function that are interesting, and it is worth understanding the logic behind it:

a. Overall, in terms of structuring, we have separated the "special cases" and the usual recursive logic. We've placed the recursive logic into its own named function (**checkPrimeCandidate**), which is shown in purple. The advantage of this approach is twofold:
    i. Separating "special" and "general" cases can help a programmer to see the structure of an algorithm more clearly.

        ii. The “special” cases will be checked *once*, instead of being checked every time that a recursive call is made. This could improve the performance of our code[1].
    b. There’s no need for **betterCheckPrime** to be recursive; it never calls itself. So we can remove the **rec** keyword from before it.
    c. Notice that on Line 1, we only bind **number**: there’s no **x** that is bound. The **x** was only needed for the *recursive logic*. We have separated that into its own function, which we call from within **betterCheckPrime**[2]. We therefore have *full control* over what we pass to it, and we do not need to rely on the user to supply a suitable “start” value.
    d. If you look at Line 2, you’ll notice that **checkPrimeCandidate** only binds <u>*one input*</u>, the **x** value, which is used as the “value to divide by”. Look at the original code, and you’ll see that *two* inputs are bound. What happened to **number**? Well, there are two things to note:
        i. Firstly, **number** is certainly available within **checkPrimeCandidate**: look at lines 3 and 5, where the symbol is used. This is because **checkPrimeCandidate** is located *within* the scope[3] of **betterCheckPrime**. The scope of the symbol is the *entire body* of **betterCheckPrime**. We do not need to separately bind a symbol that is already in scope. You will see this sort of technique being used against and again.
        ii. Secondly, if you look at the original code, you’ll see that **number** doesn’t change throughout the algorithm, but new values of **x** are generated. So there is no need to ever bind a different value to **number** in a recursive function call.

7. Let’s run some small checks, so that we can see that our revised **betterCheckPrime** is working. Copy and paste the previous tests, then modify them to refer to **betterCheckPrime**. Ensure that they’re all passing.

Simple List functions

8. We will begin by defining a custom list type. Write this in the file:

```
type MyList<'data> =
| Empty
```

[1] Or, in a well-optimised functional language, it might not make any difference at all. Remember that if a function is called with the same input, it will result in the same output. A smart compiler might identify that the same function is being called with the same input, and decide to completely eliminate the entire branch in any recursive call. This kind of optimisation is difficult and error-prone in an imperative language, but is completely safe in a functional language.
[2] See Line 12, where we pass through the value 2 as our “start” value.
[3] Remember that *scope* is determined by *indentation*, just like it is in Python! **checkPrimeCandidate** is *within* the indented body of **bettercheckPrime**, so it is within the scope of **betterCheckPrime**.

```
| Element of 'data * MyList<'data>
```

You've seen something similar in class and we demonstrated it there as well: it is a *parametrically polymorphic sum type*, which is a fancy way of saying that "it is a set of distinct cases and at least one of those cases can hold 'anything' "—but we say "parametrically polymorphic sum type" because it's a bit shorter than saying that whole sentence.

Remember that before we use this **MyList<'data>** type in a function, we will have to send it down to F# Interactive. You can do that now by selecting it and pressing Alt+Enter. Something else to remember is that the F# language is written from top-to-bottom, so everything that a function uses must already be defined above it[4].

9. In the F# Interactive REPL[5], try to create a list with nothing in it. Type **Empty** , follow it with two semicolons, and press **Enter**. You should see the output:

   ```
   val it: MyList<'a>
   ```

   You'll be using F# Interactive a lot, and you need to make sure that you're able to actually evaluate code in it. If something weird happens with your REPL, just close it (click on the trashcan icon at the top-right of the panel). **Alt+Enter** when you send code for evaluation should always open it up again.
10. Now try to create a list that contains one element. Try this input:
    Element (80, Empty)
    Once again, you should see an output. You can also try to create a list with a string in it, or a two-element list, and so on, until you are comfortable with using the REPL.
11. Now that you're comfortable with your custom list, let's make a simple function to help us play with it. Write this into your file:

    ```
    let push v l = Element (v, l)
    ```

    This creates an **Element** with the inputs. What a simple function! It will just make things more readable for us.
12. We'll now make a slightly more complicated function which reverses a list. Write this function:

    ```
    let reverse list =
      let rec reversal output =
        function
        | Empty ->
            output
    ```

[4] If you see red lines in your code and it tells you that a custom type isn't defined, maybe this is the problem.
[5] Read-Execute-Print-Loop.

```
6      | Element (x, remaining) ->
7          reversal (push x output) remaining
8    reversal Empty list
```

Verify that this evaluates correctly. What can we say about this function?

a. Once again, we see the familiar pattern of inner-function which does the real work.
b. Puzzle through the function yourself. Run it with some test values. Does it do what you expect? Can you work out how it is doing it? After coming up with your own explanation, see my explanation that follows this paragraph, and see if they are the same.

   We call **reversal** (the recursive inner function that does the real work) and pass it the list that we obtained from the input. In **reversal**, we also have an **output** symbol which is initially bound to **Empty** (see line 8). If the input-list is **Empty**, then we return the **output** symbol—which, as stated, was initially bound to **Empty**. However, if the input-list is *not* **Empty**, then we recursively call **reversal** with the remaining items and with a newly-created **output** value that is prefixed with the head that was matched. So as we see each of the *outermost* values of the list that's given to us, we generate a new list where the *innermost* values are built up from the *outermost* values of what we're given. The list we pass to **reversal** decreases in size by 1 element each time, and eventually it will be **Empty**; and when it is, we return **output**, which is our final result.

13. Of course, F# has its own built-in list type which is *exactly the same* as our **MyList<'data>** type, but has easier syntax to type. Let's create a reverse function for the built-in list:

```
let listReverse list =
  let rec reversal output =
    function
    | [] ->
        output
    | x :: remaining ->
        reversal (x :: output) remaining
  reversal [] list
```

Verify that this evaluates correctly.

14. Now have a look at **reverse** and **listReverse**. Try out **listReverse** and get comfortable with it.

    Can you see that the two are, in fact, *exactly the same* except for some syntax changes? 🤯

15. Let's make another recursive function that uses the built-in list. I've written it below, and you can type it into your file. *Without* reading the explanation that

follows, can you try to figure out—from the code—what it does? Then see if your explanation and mine are the same.

```
1  let rec listTake n l =
2    (function
3    | (0, _) → []
4    | (_, []) → []
5    | (_, h::t) → h :: (listTake (n-1) t)
6    ) (n, l)
```

...

The hint is in the name. This function will "take" the first *n* items from the supplied list. If there are fewer than *n*, then it will take as many as it can. It works by checking whether there are zero elements to take (line 3) and if so, then it returns an empty list. Likewise, if the input list is empty (line 4), the output list should be empty. Lastly, if there is something to take (line 5), then take it—and prepend it to the final output, which will contain *n*-1 items via a recursive call.

There's also a little technique here that might be quite helpful:

a. The function (line 1) accepts **n** and **l** . We actually want to use *both* of these *together* in our logic, just because it's convenient for us.
b. Lines 2-5 define a function where, indeed, **n** and **l** are pattern-matched as if they arrived in a tuple.
c. Line 6 *creates a tuple value* **(n, l)** and supplies it to the aforementioned function. This gives it to our function in the preferred format that we happen to want to use.

Trees 🌳🌴🌲

In this section, we will create a parametrically polymorphic data type `Tree<'a>` that represents a binary search tree. A binary tree is a data structure in which each element (commonly referred to as a node) has at most two children, usually distinguished as the left child and the right child. Each child node is the root of its own subtree, and this hierarchical structure continues recursively. The topmost node in the tree is called the root. The nodes without any children are called leaf nodes. The depth of a node is the number of edges from the root to the node. The height of a tree is the maximum depth of any node in the tree.

16. We will begin by defining a **Tree<'a>** type.

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

Have a read through the different cases. Can you see that they cover *all* of the possible ways in which a tree can be structured?

17. See if you can make a tree from each case, in the REPL. Perhaps your trees will look something like:

```
EmptyTree
NoChildren 5
RightOnly (5, EmptyTree)
... you can continue on from here.
```

18. One can make very simple functions for this tree, such as **left** to get the left sub-tree—if it exists—and **right** to get the right one, if it exists.

```
let left =
  function
  | EmptyTree -> EmptyTree
  | RightOnly _ -> EmptyTree
  | NoChildren _ -> EmptyTree
  | LeftOnly (_, t) -> t
  | TwoChildren (_, t, _) -> t
```

19. That looks very repetitive, but there's a small trick that one can use to combine cases in a pattern-match in the F# language.

```
let left =
  function
  | EmptyTree | RightOnly _ | NoChildren _ -> EmptyTree
  | LeftOnly (_, t) | TwoChildren (_, t, _) -> t
```

However, this trick can only be used when the *same names* are bound in all of the combined patterns.

20. You can make a **right** function similarly. Can you make it *without* looking at the code block below?

```
let right =
  function
  | EmptyTree | LeftOnly _ | NoChildren _ -> EmptyTree
  | RightOnly (_, t) | TwoChildren (_, _, t) -> t
```

21. We might also want to get all the values in the tree. Unlike a list, a tree has no "natural" order for returning nodes. We will simply obtain the values in the tree using the order of Node-Left-Right (NLR) order[6].

```
let rec treeNLR =
  function
  | EmptyTree -> []
  | NoChildren x -> [x]
```

[6] You may have seen this called "preorder traversal" in other courses.

```
  | LeftOnly (x, xs) -> [x] @ (treeNLR xs)
  | RightOnly (x, xs) -> [x] @ (treeNLR xs)
  | TwoChildren (x, xs, ys) ->
    ([x] @ (treeNLR xs)) @ (treeNLR ys)
```

Try this out; does it work as expected?

That takes us to the end of the warm-up. Hopefully, you have had plenty of practice with recursion, recursive data-structures, patterns, lists, and recursive functions by now!

Practical [20]

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.

1. [5] Create a **listDrop** function which will drop the first *n* items of a list, and return the rest. If there are fewer than *n* items, return an empty list.
2. [5] Create a **listInsert** function which, given a value **v** and a sorted list, will return a new list that contains **v** in the correct place.
3. [5] Create an **insert** function which, given a **Tree<'a>** and a value **v**, returns a **Tree<'a>** including **v** such that the following conditions hold:
    a. All left-children are less-than their parents.
    b. All right-children are greater-than their parents.
    c. There are no duplicate values (if a value is already in the tree, there is nothing that needs to be done).
4. [5] Create a **treeLNR** function which, given a **Tree<'a>**, will return the elements of the tree in a list, traversing the tree in Left-Node-Right order.

Upload your .fsx file to RUConnected.

Homework [10]

1. [5] Create a **listMerge** function that accepts a list of lists and returns a single list, with no duplicates, in sorted order.
2. [5] Create a **treeMerge** function that accepts a list of trees and returns a single tree that contains all the values in the trees.

Upload your .fsx file to RUConnected.