Practical 4

CSc302 Functional Programming

Deadline: **19 March** at **2pm**
In this practical, we will focus on the conversion of partial functions to total functions.

Warm-up

From this point on, you will be permitted to use *any built-in F# function* from the **List**, **Map**, **Set**, **Option**, **Result**, and **String**[1] namespaces as long as that function is **pure**. You can click on the links to get to the documentation, and click on the ▸ to expand documentation. Expanding the documentation will show any exceptions that a function throws. Remember that purity of a function means that both these conditions hold:

The function does not rely on anything except for its input to generate its output. (in other words: it must be *deterministic*).
The function does not have any externally-visible impact, apart from generating a value. (in other words: it cannot have any *side-effects*).

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.

The Problem with Exceptions

2. F# is a *multi-paradigm functional-first* language. This means that it supports object-orientation and functional programming, but encourages (via syntax & semantics) a functional approach. Unfortunately, this means that it allows nondeterministic functions (e.g. input functions to read input from a user) and functions with side-effects (e.g. those that throw exceptions).

3. Let's introduce you to some of these functions: **List.last** and **List.init**. The first of these functions, **List.last**, returns the last element of a list. The second one "creates a list by calling the given generator on each index". There are some examples in the documentation; let's try some out in the F# REPL:

```
List.init 4 (fun v → v + 5)
List.init 6 (fun x → $"Item #{x}")
```

[1] Note that a string is considered to be a *sequence* of characters. So, for example, you can create a list of characters from a string by using the **List.ofSeq** function.

Can you see what the output is? Play around with it until you are satisfied that you have some idea what it does. Try it with zero. Try it with a negative number. What happens in these cases?

4. Let's also play around with **List.last**. What happens when you pass it a list with 4 elements? 2 elements? 1 element? 0 elements?

5. Now, let's say that we want to make a function that
    a. accepts a list;
    b. gets the last number in the list;
    c. creates a new list of that size with the values 1...size, and returns it.

   It might look something like this:

```
let toyExample xs =
  List.init (List.last xs) (fun a → a + 1)
```

6. If I try this out with code such as **toyExample [5; 3; 4]** , I will arrive at the value **[1; 2; 3; 4]**. What's happened to get to this code?
    a. The *last element* of [5;3;4], i.e. **4**, was taken from the list.
    b. This was used as the list length, giving us **List.init 4 (fun a -> a + 1)**.
    c. That generated our list [1; 2; 3; 4].

   Play around with **toyExample** until you're familiar with it. The code for this, as you can see, is a one-liner.

7. What happens when you write **toyExample [ ]** , or **toyExample [8; 6; -5]** ? Oh no! 😱

   Can you see why it happens?

8. Since F# is impure, many functions might throw exceptions: in other words, they are *partial functions*. We will not use these functions. Instead, we will reimplement them as *total functions*. Let's start with one of the simplest of these: **List.last**. When the list is empty, it can throw an **ArgumentException**. So, let us reimplement it:

```
1  let listLast =
2    function
3    | [x] → Some x
4    | [] → None
5    | _::t → listLast t
```

9. This is now a *total* function which uses the built-in **Option<'a>** type. Instead of throwing an exception, it *expands the codomain* with a value that indicates that there is no such item as the "head of an empty list".

   In the F# language, you will often find **Option<'a>**-returning functions with a "try" prefix. For example, **List.tryLast** does the same thing as the function we

just wrote. Unfortunately, not all functions have a "try" version.

10. Let's do it with the other function, **List.init**, which doesn't have a "try" version. Can you implement it so that it becomes total? Try to do it yourself. When you're done, see how it matches up with the following implementation:

```
1  let listInit length initializer =
2    let rec doInit acc n =
3      if n = length then
4        List.rev acc
5      else
6        doInit (initializer n :: acc) (n+1)
7    if length < 0 then
8      None
9    else
10     Some (doInit [] 0)
```

11. We've used two interesting techniques here.

    The first is tail recursion in **doInit**. You've seen this in class already; this is just another example of it. We pass new values for **acc** forward to **doInit** recursively (line 6), and so we can say **doInit** is in *tail position*: there is nothing to do after it returns, except return a result. In the base case (line 4), we just reverse the list, which we built up using the very efficient "cons" operation.

    The second technique can be seen on lines 7-10. We do our check for length on line 7, and return our "error" case (**None**) on line 8. When we know that we are able to complete the problem, we create our list value on line 10 (via **doInit [] 0**) and we *wrap* it in a **Some** type. Note that as far as the **doInit** function itself is concerned, it knows nothing about **Option<'a>** at all; it's just making a list. The "wrapping" occurs right at the end, when the correct value to wrap is available.

12. Great! Now we have total functions where errors can't occur. Let's make a **toyExample** function where errors can't occur:

```
1  let toyExample xs =
2    (function
3    | None → None
4    | Some n → listInit n (fun a → a + 1)
5    ) (List.tryLast xs)
```

13. You can run the previously-failing examples and verify for yourself that nothing crashes; instead, the **None** value is returned.

    But what happened to our nice one-liner? It has disappeared! The code is much more difficult to read, too. And in fact, every time that we use an **Option<'a>**, our calling code becomes uglier 😒.

Let's see if we can do something to improve this situation.

The Solution with Option<'a>s

14. Let's start by looking at the structure of the calling code, using our "toyExample" from above.

```
1  let toyExample f xs =
2    (function
3    | None → None
4    | Some n → listInit n (fun a → a + 1)
5    ) (List.tryLast xs)
```

    The first thing to be executed is on line 5 (**List.tryLast xs**). That is passed to the function that runs between lines 2-4. If there was no last element, then **None** would have been returned, so line 3 matches and **None** is returned again. However, if there was a last element, then a **Some** value would have been returned, so line 4 matches and we execute the body **listInit n (fun a -> a + 1)**. This, in turn, will either produce a **Some** value or **None**, and that is what will be returned.

15. Let's generalise a bit. In this case, we have two functions—**tryLast** and **listInit**—that we want to "chain" together, i.e. run one-after-another.

    The second one—**listInit**—will accept an **int** , and the first one—**tryLast**—will produce an **Option<int>**. If either of them produces **None**, then the final result will be **None**.

    Intuitively, this is a little bit similar to the way that the "compose" or "pipe" operators work. Remember that with both of these, the codomain of the first function must match the domain of the second function. However, in this case, if the codomain of the first function is **Option<something>**, then the domain of the second function must be **something**.

16. Just to remind ourselves how composition and piping works, if we have two functions **f** and **g**, and their composition **h**, and we provide an input **x**, the following are all equivalent:
    a. `x ▷ f ▷ g`
    b. `f x ▷ g`
    c. `x ▷ (f >> g)`
    d. `(f >> g) x`
    e. `x ▷ h`
    f. `h x`

    Let's associate some values with the symbols, and then you can try them out yourself to see if they make sense. In the F# REPL,

```
let x = 4
let f = fun a → a + 3
let g = fun v → v * 2
let h = f >> g
```

Now that we have bound the symbols to values, you should be able to try any of the examples (a)...(f), and verify for yourself how they work. Play with the syntax as much as you want; see what works and what doesn't! Before you continue, try to have some sort of intuitive grasp of how the "pipe" and "compose" operators work.

17. Ideally, we want to achieve something like the form of (a): `x ▷ f ▷ g`. We can't have this directly, because the domain and codomain of the **f** and **g** aren't going to align directly. So when a value is being piped to **g**, we will have to *extract* the "wrapped" type from the output of **f**, and then pass it to **g**—if there is, actually, a **Some** value. If there's just a **None**, then we don't need to call **g** at all.

    But how do we do this?

    We will use a function, of course![2] Our function will be a *higher-order function*: it will accept a **g**-function and *if* the input is a **Some** value, it will run the **g** function with it. If it's a **None** value, then it'll just return None. For fun, I'll call my extra-special higher-order function "**bind**", because it *binds together* **Option<'a>**-returning functions easily.

18. Our first stab at this might look like this:

```
1  let bind f =
2    function
3    | None → None
4    | Some n → f n
```

    Let's review what this does.

    It is a function with arity 2 (it takes a function **f** and an unnamed input which must be an **Option<'a>**). If the unnamed input is **None**, then **None** is returned (line 3). If the unnamed input is a **Some** value, then we pattern-match to get that value (line 4) and we apply the function **f** with that value as an input. The result of **f** must be an **Option<'a>** type, because the result of the other branch is an **Option<'a>** type. Notice that the *input* to **f** is not an **Option<'a>** type; it is an **'a** type! As far as **f** is concerned, it doesn't even know that **bind** might have refused to invoke it with a value (if it took the line-3 path).

    Put it into the F# REPL and look at the type. Do you think, based on the type, that the function might work?

[2] Functions are the solutions to most things in functional programming.

19. Now we'll try to make a simpler **toyExample**. Recall that our *original* (exception-throwing) **toyExample** looked like this:

```
1  let toyExample xs =
2    List.init (List.last xs) (fun a → a + 1)
```

We could rewrite that with a pipe, so that it looks like this:

```
1  let toyExample xs =
2    List.last xs ▷ fun n → List.init n (fun a → a + 1)
```

Notice that I haven't actually changed anything by doing this. I've just created a lambda function that passes **n** into the right place.

If we wanted our toyExample to look a bit nicer, we could try splitting it over one more line:

```
1  let toyExample xs =
2    List.last xs
3    ▷ fun n → List.init n (fun a → a + 1)
```

Make sure you know how this works! Experiment if you need to!

20. Now let's try achieving something that's equally good-looking, but without exceptions:

```
1  let toyExample xs =
2    List.tryLast xs
3    ▷ bind (fun n → listInit n (fun a → a + 1))
```

Before trying this out: *how does it work*?

Let's review what **bind** does. It will accept a function (and on line 3, we give it a function). This function will only be called *if* the input from **tryLast** is a **Some** value, in which case, the actual value associated with the **Some** will be pattern-matched and passed to it (see explanation of **bind**, above).

Now try it out. Can you get it to fail? Or is it total?

21. Here is an insight: you can do the same trick with *any number* of **Option<'a>**-producing functions, chaining them along with pipes and just using the "bind" function to modify them to accept the actual value rather than an **Option<'a>**.

However, there's something we're missing. We can chain together a lot of **Option<'a>**-producing functions, but what happens if we want to use a function that *doesn't* produce an **Option<'a>**? For example, we might want to use a **List.filter**, but we won't be able to because it doesn't produce an **Option<'a>**. If you want to, you can try either of these (non-working) pieces of code and see for yourself:

```
listInit 5 (fun x → x * 3)
▷ List.filter (fun x → x % 2 = 0)
```

The codomain of this **listInit** application is **Option<List<int>>** and the domain of **List.filter** is **List<int>**. The two don't match, so using the first as the input to the second is not going to work.

But don't we have **bind**, which solves this particular problem for us? Yes, we do. But it won't help us; try it out:

```
listInit 5 (fun x → x * 3)
▷ bind (List.filter (fun x → x % 2 = 0))
```

Try it and see what the error message is. The problem here is that **bind** expects the *codomain* of the specialist function (i.e. **List.filter (fun x -> x % 2 = 0)**) to be an **Option<'a>**, and that's not what a **List.filter** generates. So, are we able to have total functions and no errors—but only if we give up being able to call "normal" functions easily?

22. Not at all 😄. We can have the best of both worlds. The problem is that the **bind** is accepting a specialist function which is expected to produce an **Option<'a>**. But what if we took in a "normal" function, and we used it to generate an **Option<'a>** value?

    In fact, we have a name for this kind of operation that operates in a data structure, transforms the value, and returns a new data structure with a transformed value. Can you remember what it is called? (answer is in the next paragraph)

    Of course, it is a **map**!

```
1  let map f =
2    function
3    | None → None
4    | Some n → Some (f n)
```

    This is very similar to the **bind**; in fact, only one line is changed. And it is changed to use the same trick we've seen before, when we made **listInit**: we call the function and then "wrap" it into a **Some** value.

23. Let's see how this all works together. Try this (working) code out:

```
listInit 5 (fun x → x * 3)
▷ map (List.filter (fun x → x % 2 = 0))
```

    Voilà!

24. One last thing. We might want a way to drop out of the world of **Option<'a>**s, and get back to the "normal" world of non-**Option<'a>**s. Ideally, we'd like to extract the value from the **Option<'a>** and return it.

    But what do we do if the **Option<'a>** is a **None** value?

    We have to decide, in that case, what value we want to *default* to. If there is a value in the **Option<'a>**, we will give it back; and if there isn't, we will return the specified default value.

    Fortunately, such a function (which we will call **defaultValue**) is very easy to write. Why not try to write it before you see my solution?

```
1  let defaultValue default =
2    function
3    | None → default
4    | Some v → v
```

25. Functions like **bind**, **map**, and **defaultValue** are extremely useful for error-handling in a functional language. In fact, they are so useful that the language comes with them built-in! You can find the built-in versions as **Option.bind**, **Option.map**, and **Option.defaultValue**.

If you prefer to use the **Result<'a, 'b>** type, you might be interested in the functions **Result.bind**, **Result.map**, and **Result.defaultValue**. That concludes this warm-up.

Homework [35]

1. [8] Create total versions of the following F# library functions:
    a. List.average
    b. List.averageBy
    c. List.insertAt
    d. List.maxBy

2. [17] You will have noticed that a lot of exceptions occur because it is possible for lists to be empty. Define a data structure that represents a non-empty list—in other words, a list which can *never* have zero items in it—and create **map**, **filter**,

and **fold** functions for your type.

3. [5] Write a function **copypaste** which
    a. accepts a list and three integers (**start** and **num** and **idx**);
    b. “copies” the elements starting from index **start** and “copying” a total of **num** elements;
    c. “pastes” the copied elements into the list at index **idx**;
    d. returns the new list.

    You should try to implement separate functions for steps (b) and (c). If it is impossible to do a particular operation (e.g. **start** is < 0 or **num** is greater than the length of the list or **idx** is out of bounds), then **None** should be returned.

4. [5] Write a function **makeOracle** that accepts a string and returns a function which accepts a character value and returns, if possible, the number of times that the character occurs in the string.

    Assuming that the binding **let oracle = makeOracle "Hello, world! This is me."** exists, we expect that:
    a. oracle 'H' = Some 1
    b. oracle 'h' = Some 1
    c. oracle 'e' = Some 2
    d. oracle ' ' = Some 4
    e. oracle 'z' = None

    Of course, these values would be different if **makeOracle** was passed a different string.