BFNP–F2016, Functional Programming The IT University, Spring 2016

Exercise 2

Last update 2016-01-30

This exercise sheet must be handed in via LearnIt by February 16th.

You are welcome and enquared to solve the assignments in pairs.

Your name must be part of the filename, e.g., `BFNP-02-<name1>-<name2>.fsx`, where `<name1>` and `<name2>` are the names of the two working together. Both `name1` and `name2` must upload the same file. An example: `BFNP-02-MadsAndersen-ConnieHansen.fsx`.

You can only upload one file and it must be of type `fs` or `fsx`.

It is important that you annotate your own code with comments. It is also important that you apply a functional style, i.e., no loops and no mutable variables.

For this hand-in you also need to consider scenarios where your solutions should return an error, i.e., an exception. The requirement is, that no matter what input you pass to your function that fulfils the function type, then the function should return the intended answer or an exception. It is up to you to define the exceptions and whether they should carry extra information, like error messages.

Exercise 2.1 Write a function

`downTo:int->int list`

so that `downTo n` returns the `n`-element list `[n; n-1; ...; 1]`. You must use if-then-else expressions to define the function.

Secondly define the function `downTo2` having same semantics as `downTo`. This time you must use pattern matching.

Exercise 2.2 Write a function

`removeOddIdx:int list->int list`

so that `removeOddIdx xs` removes the odd-indexed elements from the list `xs`:

```
removeOddIdx [x0; x1; x2; x3; x4; ...] = [x0; x2; x4; ...]
removeOddIdx [] = []
removeOddIdx [x0] = [x0]
```

Exercise 2.3 Write a function

`combinePair:int list->(int*int) list`

so that `combinePair xs` returns the list with elements from `xs` combined into pairs. If `xs` contains an odd number of elements, then the last element is thrown away:

```
combinePair [x1; x2; x3; x4] = [(x1,x2);(x3,x4)]
combinePair [x1; x2; x3] = [(x1,x2)]
combinePair [] = []
combinePair [x1] = []
```

Hint: Try use pattern matching.

Exercise 2.4 Solve HR, exercise 3.2.

Exercise 2.5 Solve HR, exercise 3.3.

Exercise 2.6 Solve HR, exercise 4.4

Exercise 2.7 Write a function

`explode:string->char list`

so that `explode s` returns the list of characters in `s`:

```
explode "star" = ['s';'t';'a';'r']
```

Hint: if `s` is a string then `s.ToCharArray()` returns an array of characters. You can then use `List.ofArray` to turn it into a list of characters.

Now write a function

`explode2:string->char list`

similar to `explode` except that you now have to use the string function `s.Chars` (or `.[]`), where `s` is a string. You can also make use of `s.Remove(0,1)`. The definition of `explode2` will be recursive.

Exercise 2.8 Write a function
`implode:char list->string`
so that `implode s` returns the characters concatenated into a string:

```
implode ['a';'b';'c'] = "abc"
```

Hint: Use `List.foldBack`.
Now write a function
`implodeRev:char list->string`
so that `implodeRev s` returns the characters concatenated in reverse order into a string:

```
implodeRev ['a';'b';'c'] = "cba"
```

Hint: Use `List.fold`.

Exercise 2.9 Write a function
`toUpper:string->string`
so that `toUpper s` returns the string `s` with all characters in upper case:

```
toUpper "Hej" = "HEJ"
```

Hint: Use `System.Char.ToUpper` to convert characters to upper case. You can do it in one line using `implode`, `List.map` and `explode`.
Write the same function `toUpper1` using forward function composition
`((f » g) x = g(f x))`.
Write the same function `toUpper2` using the pipe-forward operator (`|>`) and backward function composition (`«`).
Hint: `«` is defined as `(f « g) x = (f o g) x = f(g(x))`.
Hint: `|>` is defined as `x |> f = f x`.
The two operators are by default supported by F#. You can have F# interactive print the types:

```
> (<<);;
val it : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b) = <fun:it@3-4>
> (|>);;
val it : ('a -> ('a -> 'b) -> 'b) = <fun:it@4-5>
> (>>);;
val it : (('a -> 'b) -> ('b -> 'c) -> 'a -> 'c) = <fun:it@5-6>
```

Exercise 2.10 Write a function
`palindrome:string->bool`,
so that `palindrome s` returns `true` if the string `s` is a palindrome; otherwise false.
A string is called a palindrome if it is identical to the reversed string, eg, "Anna" is a palindrome but "Ann" is not. The function is not case sensitive.

Exercise 2.11 The Ackermann function is a recursive function where both value and number of mutually recursive calls grow rapidly.
Write the function
`ack:int*int->int`
that implements the Ackermann function using pattern matching on the cases of `(m,n)` as given below.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

What is the result of `ack(3,11)`.
Notice: The Ackermann function is defined for non negative numbers only.

Exercise 2.12 The function
`time f:(unit->'a)->'a*TimeSpan`
below times the computation of `f x` and returns the result and the real time used for the computation.

```
let time f =
  let start = System.DateTime.Now in
  let res = f () in
  let finish = System.DateTime.Now in
  (res, finish - start);
```

Try compute `time (fun () -> ack (3,11))`.
Write a new function

`timeArg1 f a : ('a -> 'b) -> 'a -> 'b * TimeSpan`

that times the computation of evaluating the function `f` with argument `a`. Try `timeArg1 ack (3,11)`.
Hint: You can use the function `time` above if you hide `f a` in a lambda (function).

Exercise 2.13 HR exercise 5.4