Learning to Program with F#
Exercises
Department of Computer Science
University of Copenhagen

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

October 21, 2022

0.1 Lists

0.1.1 Teacher's guide

Emne lister og arrays

Sværhedsgrad Middel

0.1.2 Introduction

A list is a very important programming data structure, and functional programming is particularly well suited for processing lists. Hence, F# has constructs that support list operations, and some of these will be worked with in the following assignments.

0.1.3 Exercise(s)

0.1.3.1: Use `Array.init` to make a function `squares: n:int -> int []`, such that the call `squares n` returns the array of the first n square numbers. For example, `squares 5` should return the array `[|1; 4; 9; 16; 25|]`.

0.1.3.2: Arrays are an alternative data structure for tables.

(a) Use `Array2D.init`, `Array2D.length1` and `Array2D.length2` to make the function `transposeArr : 'a [,] -> 'a [,]` which transposes the elements in input.

(b) Make a whitebox test of `transposeArr`.

(c) Comparing this implementation with Assignment 14d, what are the advantages and disadvantages of each of these implementations?

(d) For the application of tables, which of lists and arrays are better programmed using the imperative paradigm and using the functional paradigm and why?

0.1.3.3: The function `List.allPairs: 'a list -> 'b list -> ('a * 'b) list`, takes two lists and produces a list of all possible pairs. For example,

```
> List.allPairs [1..3] ['a'..'d'];;
val it: (int * char) list =
  [(1, 'a'); (1, 'b'); (1, 'c'); (1, 'd'); (2, 'a'); (2, 'b');
   (2, 'c'); (2, 'd'); (3, 'a'); (3, 'b'); (3, 'c'); (3, 'd')]
```

Make your own implementation using two `List.map` and `List.concat`.

0.1.3.4: Project Euler.net 1: Multiples of 3 and 5

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

0.1.3.5: Project Euler.net 2: Even Fibonacci numbers

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

0.1.3.6: Project Euler.net 3: Largest prime factor

The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number 600851475143 ?

0.1.3.7: Make a function `even: int -> bool` which returns `true` if the input is even and false otherwise. Use `List.filter` and `even` to make another function `filterEven : int list -> int list`, which returns all the even numbers of a given list.

0.1.3.8: Write the types for the functions `List.filter` and `List.foldBack`.

0.1.3.9: Write a function `printLstAlt: 'a list -> ()`, which uses `for-in`, to print every element of a given list to the screen. Ensure that your function works for lists of various types, e.g., int list and string list.

0.1.3.10: Make a function `avg: (lst: float list) -> float` using `List.fold` and `lst.Length` which calculates the average value of the elements of `lst`.

0.1.3.11: Use `Array2D.init`, `Array2D.length1` and `Array2D.length2` to make the function `transpose : 'a [,] -> 'a [,]` which transposes the elements in input. Comparing this implementation with Assignment 14d, what are the advantages and disadvantages of each of these implementations?

0.1.3.12: In the following, you are to work with different ways to create a list:

(a) Make an empty list, and bind it with the name `lst`.

(b) Create a second list `lst2`, which prepends the string `"F#"` to `lst` using the cons operator `::`. Consider whether the types of the old and new list are the same.

(c) Create a third list `lst3` which consists of 3 identical elements `"Hello"`, and which is created with `List.init` and the anonymous function `fun i -> "Hello"`.

(d) Create a fourth list `lst4` which is a concatenation of `lst2` and `lst3` using “`@`”.

(e) Create a fifth list `lst5` as `[1; 2; 3]` using `List.init`

(f) Write a recursive function `oneToN : n:int -> int list` which uses the concatenation operator, “`@`”, and returns the list of integers `[1; 2; ...; n]`. Consider whether it would be easy to create this list using the “`::`” operator.

(g) Write a recursive function `oneToNRev : n:int -> int list` which uses the cons operator, “`::`”, and returns the list of integers `[n; ...; 2; 1]`. Consider whether it would be easy to create this list using the “`@`” operator.

0.1.3.13: Write a function `printLst: 'a list -> ()`, which uses `List.iter`, an anymous function, and `printfn "%A"` to print every element of a given list to the screen. Ensure that your function works for lists of various types, e.g., int list and string list.

0.1.3.14: Sometimes tables have to be reshaped into single lists and back again.

(a) Define a function `flatten : 'a list list -> 'a list`, which concatenates a list of lists to a single list. For example, `flatten [[1; 2; 3]; [4; 5; 61]]` should return `[1; 2; 3; 4; 5; 6]`.

(b) Define the inverse function `reshape : m : int -> 'a list -> 'a list list`, which takes a number of rows and a list and returns the corresponding table. For example, `reshape 2 [1; 2; 3; 4; 5; 6]` should return `[[1; 2; 3]; [4; 5; 61]]`.

(c) Make a whitebox test of the above functions.

0.1.3.15: Use `List.map` write a function, which takes a list of integers and returns the list of floats where each element has been divided by `2.0`. For example, if the function is given the input `[1; 2; 3]`, then it should return `[0.5; 1.0; 1.5]`.

0.1.3.16: Use `List.map` to make a function `applylist : ('a -> 'b) list -> 'a -> 'b list`, which applies a list of functions to the same element and returns a list of results. For example `applylist [cos; sin; log; exp] 3.5` should return approximately `[-0.94; -0.35; 1.25; 33.11]`.

0.1.3.17: Write a function `multiplicity: x:int -> xs:int list -> int`, which counts the number of occurrences of the number `x` in the list `xs` using `List.filter`, an anonymous function, and the `Length` property.

0.1.3.18: Write a recursive function `oneToN : n:int -> int list` which uses the cons operator, `::`, and returns the list of integers `[1; 2; ...; n]`.

0.1.3.19: Write a recursive function `rev: 'a list -> 'a list`, which uses the concatenation operator `"@"` to reverse the elements in a list.

0.1.3.20: Define a function `reverseApply : 'a -> ('a -> 'b) -> 'b`, such that `reverseApply x f` returns the result of `f x`.

0.1.3.21: Write a function `reverseArray : arr:'a [] -> 'a []` using `Array.init` and `Array.length` which returns an array with the elements in the opposite order of `arr`. For eksample, `printfn "%A" (reverseArray [|1..5|])` should write `[|5; 4; 3; 2; 1|]` to the screen.

0.1.3.22: Write the function `reverseArrayD : arr:'a [] -> unit`, which reverses the order of the values in `arr` using a while-loop to overwrite its elements. For example, the program

```
let aa = [|1..5|]
reverseArrayD aa
printfn "%A" aa
```

should output `[|5; 4; 3; 2; 1|]`.

0.1.3.23: Write a function `rev: 'a list -> 'a list`, which uses `List.fold`, an anymous function, and the `"::"` operator to reverse the elements in a list. Ensure that your function works for lists of various types, e.g., int list and string list.

0.1.3.24: En snedig programmør definerer en sorteringsfunktion med definitionen `ssort xs = Set.toList (Set.ofList xs)`. For eksempel giver `ssort [4; 3; 7; 2]` resultatet `[2; 3; 4; 7]`. Diskutér, om programmøren faktisk er så snedig, som han tror.

0.1.3.25: Write a function `split: xs:int list -> (xs1: int list) * (xs2: int list)` which separates the list `xs` into two and returns the result as a tuple where all the elements with even index is in the first element and the rest in the second. For example, `split [x0; x1; x2; x3; x4]` should return `([x0; x2; x4], [x1; x3])`.

0.1.3.26: A table can be represented as a non-empty list of equally long lists, for example, the list `[[1; 2; 3]; [4; 5; 6]]` represents the table:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

(a) Make a function `isTable : llst:'a list list -> bool`, which determines whether `llst` is a legal non-empty list, i.e., that

- there is at least one element, and
- all lists in the outer list has equal length.

(b) Make a function `firstColumn : llst:'a list list -> 'a list` which takes a list of lists and returns the list of first elements in the inner lists. For example, `firstColumn [[1; 2; 3]; [4; 5; 6]]` should return `[1; 4]`. If any of the lists are empty, then the function must return the empty list of integers `[] : int list`.

(c) Make a function `dropFirstColumn : llst:'a list list -> 'a list list` which takes a list of lists and returns the list of lists where the first element in each inner list is removed. For example, `dropFirstColumn [[1; 2; 3]; [4; 5; 6]]` should return `[[2; 3]; [5; 6]]`. Ensure that your function fails gracefully, if there is no first elements to be removed.

(d) Make a function `transposeLstLst : llst:'a list list -> 'a list list` which transposes a table implemented as a list of lists, that is, an element that previously was at `a.[i,j]` should afterwards be at `a.[j,i]`. For example, `transposeLstLst [[1; 2; 3]; [4; 5; 6]]` should return `[[1; 4]; [2; 5]; [3; 6]]`. Ensure that your function fails gracefully. Note that `transposeLstLst (transposeLstLst t) = t` when `t` is a table as list of lists. Hint: the functions `firstColumn` and `dropFirstColumn` may be useful.

(e) Make a whitebox test of the above functions.

0.1.3.27: Explain the difference between the types `int -> (int -> int)` and `(int -> int) -> int`, and give an example of a function of each type.