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 Recursion

0.1.1 Teacher's guide

Emne Rekursion

Sværhedsgrad Middel

0.1.2 Introduction

Recursion is a central concept in functional programming, and is usually used instead of `for` and `while` loops. The following exercise thus train the use of recursion.

0.1.3 Exercise(s)

0.1.3.1: Use `List.fold` and `List.foldback` and your own implementations from Assignment 5 to calculate the sum of the elements in `[0 .. n]`, where `n` is a large number.

0.1.3.2: The Collatz conjecture is one of the most famous unsolved problems in mathematics[1].

$$\text{collatz(n)} \begin{cases} \frac{n}{2} & \text{if } n\%2 = 0 \\ 3n+1 & \text{if } n\%2 = 1 \end{cases}$$

(a) Implement a non-recursive function `collatzStep : n:int -> int` that gives the next number in the collatz sequence. Examples:

```
collatzStep 1 = 1
collatzStep 2 = 1
collatzStep 3 = 10
collatzStep 9 = 28
```

The function should use pattern matching.

(b) Implement a recursive function that counts the number of steps in the collatz sequence starting at n:

```
collatzStepsHelper : count:int -> n:int -> int:
```

The function should use pattern matching, recursion and the previous function `collatzStep` to compute each intermediate step of a sequence.

(c) Implement a non-recursive function `collatzSteps : n:int -> int` that simply calls `collatzStepsHelper` with an initial `count` of 0.

0.1.3.3: Write a function, `fac : n:int -> int`, which calculates the faculty function $n! = \prod_{i=1}^{n} i$ using recursion.

[1] `https://en.wikipedia.org/wiki/Collatz_conjecture`

0.1.3.4: Consider the factorial-function,

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n \tag{1}$$

(a) Write a function

```
fac : n:int -> int
```

which uses recursion to calculate the factorial-function as (1).

(b) Write a program, which asks the user to enter the number `n` using the keyboard, and which writes the result of `fac n`.

(c) Make a new version,

```
fac64 : n:int64 -> int64
```

which uses `int64` instead of `int` to calculate the factorial-function. What are the largest values n, for which `fac` and `fac64` respectively can correctly calculate the factorial-function for?

0.1.3.5: Make your own implementation of `List.fold` and `List.foldback` using recursion.

0.1.3.6: The greatest common demoninator of 2 integers t and n is the greatest number c, which integer-divides both t and n with 0 remainder. Euclid's algorithm[2] finds this greatest common denominator via recursion:

$$\gcd(t,0) = t, \tag{2}$$
$$\gcd(t,n) = \gcd(n, t \ \% \ n), \tag{3}$$

where % is the remainder operator (as in F#).

(a) Implement Euclid's algorithm as the recursive function:

```
gcd : int -> int -> int
```

(b) Test your implementation.

(c) Make a tracing-by-hand on paper for `gcd 8 2` and `gcd 2 8`.

0.1.3.7: Write a function `lastFloat : float list -> float` that, using recursion, returns the last element of the argument list if the list is non-empty and returns the float value `NaN` if the argument list is empty.

For example, the call `lastFloat [2.1;4.2]` should return the float value `4.2` and the call `lastFloat []` should return the value `NaN`.

0.1.3.8: Write a function `length : 'a list -> int` that calculates the length of the argument list using recursion. The function should make use of pattern matching on lists.

0.1.3.9: Write a tail-recursive function `lengthAcc : acc:int -> xs:'a list -> int` that calculates the sum of `acc` plus the length of `xs` using tail-recursion. For instance, a call `lengthAcc 0 xs` should return the length of `xs`, and a call `lengthAcc n []` should return the value `n`.

[2] `https://en.wikipedia.org/wiki/Greatest_common_divisor`

0.1.3.10: Write a function `map : ('a -> 'b) -> 'a list -> 'b list` that takes a function and a list as arguments and returns a new list of the same length as the argument list with elements obtained by applying the supplied function on each of the elements of the argument list. The function should make use of recursion and pattern matching on lists.

As an example, a call `map (fun x -> x+2) [2;3]` should return the list `[4;5]`.

0.1.3.11: Write a function `pow2 : n:int -> int` that calculates the value 2^n using recursion. The function should return the value 0 when $n < 0$.

0.1.3.12: Using Steps 1, 3, 5, 7, and 8 from the 8-step guide

(a) write a recursive function which takes two integer arguments x and n and returns the value x^n.

(b) write another function wich takes one argument (x,n) and calls the former.

Document both functions using the `<summary>`, `<param>`, and `<returns>` XML tags. Consider what should happen, if $n < 0$, and whether there is any significant difference between the call of the two functions.

0.1.3.13: Write a function, `sum : n:int -> int`, which calculates the sum $\sum_{i=1}^{n} i$ using recursion. Make an implementation using a `while` loop, compute a table for the values $n = 1..10$ and compare the results. Discuss the difference between the two methods.

0.1.3.14: The following code calculates the sum $\sum_{i=0}^{n} i$ using recursion and 64-bit unsigned integeres.

```
let rec sum (n: uint64) : uint64 =
  match n with
    0UL -> 0UL
    | _ -> n+sum (n-1UL)

let n = uint64 1e4
printfn "%A" (sum n)
```

Depending on the computer, this program runs out of memory for large values of n, e.g., not many compupters can run this program when `n = uint64 1e10`. The problem is, that the algorithm is not using tail recursion.

(a) Explain why this is not tail recursion.

(b) Run the code on your computer and empirically find a small n for which the program runs out of memory on your computer.

(c) Rewrite the algorithm to become tail recursive by including the accumulated sum as the argument:

```
sum (acc: uint64) -> (n: uint64) -> uint64
```

and check that this does not run out of memory for the above identified n. Try also bigger.

(d) For which values of n would the tail recursive version break and why?

0.1.3.15: Write a function `sum : int list -> int` that takes a list of integers and returns their sum. The function must traverse the list using recursion and pattern matching.

0.1.3.16: Consider the following sum of integers,

$$\sum_{i=1}^{n} i \tag{4}$$

This assignment has the following sub-assignments:

(a) Write a function

```
sum : n:int -> int
```

which uses pattern-matching and recursion to compute the sum $1+2+\cdots+n$ also written in (4). If the function is called with any value smaller than 1, then it is to return the value 0.

(b) By induction one can show that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, n \geq 0 \tag{5}$$

Make a function

```
simpleSum : n:int -> int
```

which uses (5) to calculate $1+2+\cdots+n$ and which includes a comment explaining how the expression implemented is related to the mentioned sum.

(c) Write a program, which asks the user for the number n, reads the number from the keyboard, and write the result of `sum n` and `simpleSum n` to the screen.

(d) Make a program, which writes a table to the screen with 3 columns: `n`, `sum n` and `simpleSum n`. The table should have a row for each of $n = 1,2,3,..,10$, and each field must be 4 characters wide. Verify programmatically that the two functions calculate identical results.

(e) What is the largest value n that the two sum-functions can correctly calculate the value of? Can the functions be modified, such that they can correctly calculate the sum for larger values of n?