F sharp Master Document

Created	@June 3, 2022 8:58 AM
Tags	4. Semester

Course plan

Week number	Functional Programming	Second-year project
5		
6	Assignment 1 [Feb 8]	
7	Assignment 2 [Feb 15]	
8	Assignment 3 [Feb 22]	
9	Assignment 4 [Mar 1]	
10	Assignment 5 [Mar 8]	
11	Assignment 6 [Mar 15]	
12		
13	Assignment 7 [Mar 29]	Feedback session 1
14	Project start	
15	Easter Vacation	Easter Vacation
16		
17		
18		Feedback session2
19		
20	Project submission	
21		
22	Exam [Jun 3]	Tech report submission [May 30]
23		Exam [Jun 7-10]
24		Exam [Jun 13-14]
25		Exam [Jun 20-23]

💡 Go through all the assignments, insert correct answers in **Assignments** tab

Go through all exams, insert correct answers in
Exams tab

If needed write notes about specific topics under
Notes tab

```
dotnet new console -lang "F#" -o ExamPractice
```


Exams

Exam 2021-08-19

Question 1

vores svar

```
module Exam2021_2
(* If you are importing this into F# interactive then
   the line above and remove the comment for the line

   Do note that the project will not compile if you do
   it does allow you to work in interactive mode and y
   to make the project compile again.

   You will also need to load JParsec.fs. Do this by t
```

```
    #load "JParsec.fs"
    in the interactive environment. You may need the en

    Do not remove the module declaration (even though t
    introduce indentation errors in your code that may
    to switch back to project mode.

    Alternative, keep the module declaration as is, but
    *)
(*
 module Exam2021_2 =
 *)

(* 1: Binary lists *)

(* Question 1.1 *)

     type binList<'a, 'b> =
     | Nil
     | Cons1 of 'a * binList<'a, 'b>
     | Cons2 of 'b * binList<'a, 'b>

     let rec length lst =
         match lst with
         | Nil -> 0
         | Cons1 (_, b) | Cons2 (_, b) -> 1 + length b

(* Question 1.2 *)
     let split lst =
         let rec aux lst' acc1 acc2 =
             match lst' with
             | Nil -> (List.rev acc1, List.rev acc2)
             | Cons1 (a, b) -> aux b (a::acc1) acc2
             | Cons2 (a, b) -> aux b acc1 (a::acc2)
         aux lst [] []
```

```
    let length2 lst =
        let rec aux lst' acc1 acc2 =
            match lst' with
            | Nil -> acc1, acc2
            | Cons1 (_, b) -> aux b (1 + acc1) acc2
            | Cons2 (_, b) -> aux b acc1 (1 + acc2)
        aux lst 0 0

(* Question 1.3 *)



    let rec map f g lst =
        match lst with
        | Nil -> Nil
        | Cons1 (a, b) -> Cons1 (f a, map f g b)
        | Cons2 (a, b) -> Cons2 (g a, map f g b)

(* Question 1.4 *)

    let rec filter f g lst =
        match lst with
        | Nil -> Nil
        | Cons1 (a, b) when f a -> Cons1 (a, filter f
        | Cons2 (a, b) when g a -> Cons2 (a, filter f
        | Cons1(_, b) | Cons2 (_, b) -> filter f g b
```

Jespers svar

```
module Question1

    type binList<'a, 'b> =
        | Nil
        | Cons1 of 'a * binList<'a, 'b>
        | Cons2 of 'b * binList<'a, 'b>

    let rec length =
```

```
        function
        | Nil -> 0
        | Cons1 (_, xs) -> 1 + length xs
        | Cons2 (_, xs)  -> 1 + length xs

    let lengthAcc lst =
        let rec aux acc =
            function
            | Nil -> acc
            | Cons1 (_, xs) -> aux (acc + 1) xs
            | Cons2 (_, xs) -> aux (acc + 1) xs

        aux 0 lst

    let length2 lst =
        let rec aux ((l1, l2) as acc) =
            function
            | Nil -> acc
            | Cons1 (_, xs) -> aux (l1 + 1, l2) xs
            | Cons2 (_, xs) -> aux (l1, 1 + l2) xs

        aux (0, 0) lst

    let rec split =
        function
        | Nil -> ([], [])
        | Cons1 (x, xs) ->
            let (r1, r2) = split xs
            (x::r1, r2)
        | Cons2 (y, ys) ->
            let (r1, r2) = split ys
            (r1, y :: r2)

    let rec map f g =
        function
        | Nil -> Nil
```

```
        | Cons1(x, xs) -> Cons1 (f x, map f g xs)
        | Cons2(x, xs) -> Cons2 (g x, map f g xs)

    let rec filter f g =
        function
        | Nil -> Nil
        | Cons1 (x, xs) when f x -> Cons1(x, filter f
        | Cons1 (x, xs)          -> filter f g xs
        | Cons2 (x, xs) when g x -> Cons2(x, filter f
        | Cons2 (x, xs)          -> filter f g xs

    let rec fold f g acc =
        function
        | Nil -> acc
        | Cons1 (x, xs) -> fold f g (f acc x) xs
        | Cons2 (x, xs) -> fold f g (g acc x) xs
```

▼ Question 2

▼ vores svar

```
(* 2: Code Comprehension *)
    let rec foo xs ys =
      match xs, ys with
      | [], ys -> ys
      | xs, [] -> xs
      | x :: xs, y :: ys when x < y ->
        x :: (foo xs (y :: ys))
      | x :: xs, y :: ys ->
        y :: (foo (x :: xs) ys)

    and bar =
      function
      | [] -> []
      | [x] -> [x]
      | xs ->
        let (a, b) = List.splitAt (List.length xs / 2)
```

```
        foo (bar a) (bar b)

(* Question 2.1 *)

    (*

    Q: What are the types of functions foo and bar?

    A:
        foo: 'a list -> 'a list -> 'a list
        bar: 'a list -> 'a list


    Q: What does the function bar do.
       Focus on what it does rather than how it does i

    A: Sorts a list using mergesort

    Q: What would be appropriate names for functions
       foo and bar?

    A:
        bar: split
        foo: mergesort

    Q: What would be appropriate names of the values a


    A:
        a: left
        b: right

    *)


(* Question 2.2 *)
```

```
    (*
    The code includes the keyword "and".


    Q: What function does this keyword serve in genera
       (why would you use "and" when writing any progr

    A: mutual recursion, function know about each othe
       would not know about it.


    Q: What would happen if you removed it from this p
       replaced it with a standard "let"
       (change the line "and bar = " to "let rec bar =
       Explain why the program either does or does not

    A: It should work, since foo is compiled earlier i

    *)

(* Question 2.3 *)
    let foo2 xs ys =
        List.unfold (
            fun state ->
                match state with
                | [], y::ys -> Some (y, ([], ys))
                | x::xs, [] -> Some (x, (xs, []))
                | x::xs, y::ys when x < y -> Some (x,
                | x::xs, y::ys -> Some (y, (x::xs, ys)
                | _ -> None
                ) (xs, ys)

    (* use the following code as a starting template
    let foo2 xs ys = List.unfold <a function goes here
```

```
    *)

(* Question 2.4 *)

    (*

    Q: Neither foo nor bar is tail recursive. Pick one
       To make a compelling argument you should evalua
       similarly to what is done in Chapter 1.4 of HR,
       You need to make clear what aspects of the eval
       Keep in mind that all steps in an evaluation ch
       ((5 + 4) * 3 --> 9 * 3 --> 27, for instance).

    A: <Your answer goes here>

    *)
(* Question 2.5 *)

    let fooTail xs ys =
        let rec aux xs' ys' c =
            match xs', ys' with
            | [], ys -> c ys
            | xs, [] -> c xs
            | x :: xs, y :: ys when x < y -> aux xs (y
            | x :: xs, y :: ys -> aux (x::xs) ys (fun
        aux xs ys id
```

jesper svar

```
module Question2

    let rec foo xs ys =
        match xs, ys with
        | [], ys -> ys
        | xs, [] -> xs
        | x :: xs, y :: ys when x < y ->
```

```
            x :: (foo xs (y :: ys))
        | x :: xs, y :: ys ->
            y :: (foo (x :: xs) ys)

    and bar =
        function
        | [] -> []
        | [x] -> [x]
        | xs ->
            let (a, b) = List.splitAt (List.length xs
            foo (bar a) (bar b)

(* Question 2.1 *)

    (*

    Q: What are the types of functions foo and bar?

    A: foo has type 'a list -> 'a list -> 'a list when
       bar has type 'a list -> 'a list when 'a: compar

    Q: What does the function bar do.
       Focus on what it does rather than how it does i

    A: Given a list lst, bar returns a sorted list in

    Q: What would be appropriate names for functions
       foo and bar?

    A: foo can be called merge
       bar can be called mergeSort

    Q: What would be appropriate names of the values a


    A: a can be called firstHalf
```

```
        b can be called secondHalf

    *)


(* Question 2.2 *)


    (*
    The code includes the keyword "and".


    Q: What function does this keyword serve in genera
       (why would you use "and" when writing any progr

    A: The keyword 'and' is used to declare mutually r
       Normally, terms are not available before they a
       this by using the 'and' keyword.

    Q: What would happen if you removed it from this p
       replaced it with a standard "let"
       (change the line "and bar = " to "let rec bar =
       Explain why the program either does or does not

    A: Nothing would happen. The program would still w

    *)

(* Question 2.3 *)
    let foo2 xs ys =
        List.unfold
            (function
             | [], []                      -> None
             | [], y :: ys                 -> Some (y,
             | x :: xs, []                 -> Some (x,
             | x :: xs, y :: ys when x < y -> Some (x,
```

```
            | x :: xs, y :: ys            -> Some (y,
           (xs, ys)

    (* use the following code as a starting template
    let foo2 xs ys = List.unfold <a function goes here
    *)

(* Question 2.4 *)

    (*

    Q: Neither foo nor bar is tail recursive. Pick one
       To make a compelling argument you should evalua
       similarly to what is done in Chapter 1.4 of HR,
       You need to make clear what aspects of the eval
       Keep in mind that all steps in an evaluation ch
       ((5 + 4) * 3 --> 9 * 3 --> 27, for instance).

    A:
        Im providing the results for both here, but re

        foo [1; 3] [2; 4; 5] -->
        1 :: foo [3] [2; 4; 5] -->
        1 :: 2 :: foo [3] [4; 5] -->
        1 :: 2 :: 3 :: foo [] [4; 5] -->
        1 :: 2 :: 3 :: [4; 5] -->
        [1; 2; 3; 4; 5]

        foo is not tail recursive as the elements are
        ( 1 :: 2 :: foo [3] [4; 5]) for instance. Unti
        append the elements 1 and 2 to, they get store
        are big enough.

        bar [8; 7; 6; 5; 4; 3; 2; 1] -->
        foo (bar [8; 7; 6; 5])
            (bar [4; 3; 2; 1]) -->
```

```
        foo (foo (bar [8; 7])
                 (bar [6; 5]))
            (foo (bar [4; 3])
                 (bar [2; 1])) -->
        foo (foo (foo (bar [8]) (bar [7]))
                 (foo (bar [6]) (bar [5])))
            (foo (foo (bar [4]) (bar [3]))
                 (foo (bar [2]) (bar [1]))) -->
        foo (foo (foo [8] [7])
                 (foo [6] [5]))
            (foo (foo [4] [3])
                 (foo [2] [1])) -->
        foo (foo [7; 8] [5; 6])
            (foo [3; 4] [1; 2]) -->
        foo [5; 6; 7; 8] [1; 2; 3; 4] -->
        [1; 2; 3; 4; 5; 6; 7; 8]


    You could absolutely get away with a shorter list

    bar is not tail recursive as it recursively appear
    bar has returned a result. These nested calls to f
    *)
(* Question 2.5 *)




    let fooTail xs ys =
        let rec aux cont xs ys =
            match xs, ys with
            | [], ys -> cont ys
            | xs, [] -> cont xs
            | x :: xs, y :: ys when x < y ->
                aux (fun result -> cont (x :: result)
            | x :: xs, y :: ys ->
```

```
            aux (fun result -> cont (y :: result))

        aux id xs ys
```

Question 3

Vores svar

```
(* Question 3.1 *)

    let nearestPerfectSquare x =
        let rec aux index prevDistance =
            let perfectSquare = index * index
            let currentDistance = abs (x - perfectSqua
            if currentDistance < prevDistance
                then aux (index + 1) currentDistance
            else
                index-1
        aux 0 1000000

    let approxSquare x n : float =
        let perfectSquare = nearestPerfectSquare x
        if n = 0
            then float perfectSquare
        else
            let rec aux (r: float) i =
                match i with
                | i when i = n -> (float) r
                | i -> aux ((((float) x / r) + r) / 2
            aux (float perfectSquare) 0

(* Question 3.2 *)

    let quadratic (a: int) (b: int) (c: int) num : (fl
        let d = (b * b) - 4 * a * c
        let sqr = approxSquare d num
        let (x, y) = (((float -b + sqr) / (2.0 * float
```

```
        (x, y)

(* Question 3.3 *)

    let parQuadratic (eqs : (int * int * int) list) (n
        List.splitInto numProcesses eqs
            |> List.map (
                fun eqs ->
                    async {
                        return List.fold (fun acc (a,b
                    }
            )
            |> Async.Parallel
            |> Async.RunSynchronously
            |> Array.fold (fun acc r -> acc@r) []
(* Question 3.4 *)

    open JParsec
    open JParsec.TextParser

    let whitespaceChar = satisfy System.Char.IsWhiteSp
    let spaces = many whitespaceChar

    let (.>*>.) p1 p2 = p1 .>> spaces .>>. p2
    let (.>*>) p1 p2  = p1 .>> spaces .>> p2
    let (>*>.) p1 p2  = p1 .>> spaces >>. p2
    let operator = pchar '+' <|> pchar '-'

    let solveQuadratic str num =
        let str' = str + "\n"
        let parser =
            pint32 .>> pstring "x^2"
            .>*>. operator
            .>*>. pint32 .>> pstring "x"
            .>*>. operator
            .>*>. pint32
```

```
            .>*> pchar '=' .>*> pchar '0'
            .>> pstring "\n"
        let result = run parser str
        let ((((a, op1), b), op2), c) = getSuccess res
        let b' = if op1 = '-' then -b else b
        let c' = if op2 = '-' then -c else c
        quadratic a b' c' num
```

Jespers svar

```
module Question3

    open JParsec.TextParser
    let area numSides length =
        System.Math.PI / float numSides |>
        System.Math.Tan |>
        (fun angle -> (length / 2.0) / angle) |>
        (fun b -> (b * (float numSides / 2.0) / 2.0))



    let closestSquare x =
        let rec aux a =
            let bsq = a * a
            if bsq >= x then
                let asq = (a - 1) * (a - 1)
                if System.Math.Abs (x - asq) <= Syster
            else
                aux (a + 1)

        aux 0

    let approxSquare a =
        let af = float a
        let rec aux b =
            function
            | 0 -> b
```

```
            | x -> aux ((b + (af / b)) / 2.0) (x - 1)

        aux (float (closestSquare a))


    let hypotenuse a b x = approxSquare (a * a + b * b

    let quadratic a b c x =
        (-(float b) + (approxSquare (b * b - 4 * a * c
        (-(float b) - (approxSquare (b * b - 4 * a * c

    let parQuadratic eqs numProcesses x =
        eqs |>
        List.splitInto numProcesses |>
        List.map (fun eqs' -> async {return List.map (
        Async.Parallel |>
        Async.RunSynchronously |>
        Array.toList |>
        List.collect id



    let pspaces = many (pchar ' ')
    let parseNum =
        pspaces >>. ((pchar '+' >>. pspaces >>. pint32
                     (pchar '-' >>. pspaces >>. pint32
    let parseQuadratic x =
        pint32 .>> pstring "x^2" .>>. parseNum .>> pch
        pspaces .>> pchar '=' .>> pspaces .>> pstring
        (fun ((a, b), c) -> quadratic a b c x)

    let solveQuadratic str x =
        str + "\n" |>
        run (parseQuadratic x) |>
        getSuccess
```

▼ Question 4

▼ Vores svar

```
(* Question 4.1 *)

    type rat = (int * int)

(* Question 4.2 *)

    let rec gcd x y =
        if y = 0 then x
        else gcd y (x % y)

    let mkRat n d =
        let x = gcd n d
        match (n / x, d / x) with
        | _, 0 -> None
        | (a', b') when a' < 0 && b' < 0 -> Some (-a',
        | (a', b') when a' < 0 || b' < 0 -> Some (-a',
        | (a', b') -> Some (a', b')

    let ratToString ((a:int), (b:int)) =
        string a + " / " + string b

(* Question 4.3 *)

    let plus (a, b) (c, d) = mkRat ((a*d)+(b*c)) (b*d
    let minus (a, b) (c, d) = mkRat ((a*d)-(b*c)) (b*
    let mult (a, b) (c, d) = mkRat (a*c) (b*d)
    let div (a, b) (c, d) = mkRat (a*d) (b*c)

(* Question 4.4 *)

    type SM<'a> = SM of (rat -> ('a * rat) option)
    let ret x = SM (fun st -> Some (x, st))
    let bind (SM m) f =
```

```
        SM (fun st ->
            match m st with
            | None -> None
            | Some (x, st') ->
                let (SM g) = f x
                g st')

    let (>>=) m f = bind m f
    let (>>>=) m n = m >>= (fun () -> n)
    let evalSM (SM f) s = f s

    let smPlus rat =
        SM (fun state ->
                match plus state rat with
                | None -> None
                | Some rat -> Some((), rat)
            )

    let smMinus rat =
        SM (fun state ->
                match minus state rat with
                | None -> None
                | Some rat -> Some((), rat)
            )

    let smMult rat =
        SM (fun state ->
                match mult state rat with
                | None -> None
                | Some rat -> Some((), rat)
            )

    let smDiv rat =
        SM (fun state ->
                match div state rat with
                | None -> None
```

```
                  | Some rat -> Some((), rat)
              )
(* Question 4.5 *)

    (* You may solve this exercise either using monadi
        using computational expressions. *)

    type StateBuilder() =

        member this.Bind(x, f)    = bind x f
        member this.Zero ()       = ret ()
        member this.Return(x)     = ret x
        member this.ReturnFrom(x) = x
        member this.Combine(a, b) = a >>= (fun _ -> b)

    let state = new StateBuilder()

    let rec calculate (opList:(rat * (rat -> SM<unit>)
        state {
             match opList with
             | [] -> return ()
             | (rat, f)::opList ->
                 do! f rat
                 return! calculate opList
        }
```

Jesper svar

```
module Question4

    type rat = R of int * int

    let gcd (a : int) (b : int) =

        let max =
            if a = 0 then System.Math.Abs b else
```

```
        if b = 0 then System.Math.Abs a else
        min (System.Math.Abs a) (System.Math.Abs b
    let rec aux acc =
        function
        | x when x > max -> acc
        | x when a % x = 0 && b % x = 0 -> aux x (
        | x                             -> aux acc

    aux 1 1

let mkRat a =
    function
    | 0 -> None
    | b -> let d = gcd a b
           if b < 0 then
               Some (R (-a / d, -b / d))
           else
               Some (R (a / d, b / d))
let ratToString (R (n, d)) = sprintf "%d / %d" n d

let plus (R (a, b)) (R (c, d)) = mkRat (a * d + c
let minus (R (a, b)) (R (c, d)) = mkRat (a * d - c

let mult (R (a, b)) (R (c, d)) = mkRat (a * c) (b

let div (R (a, b)) (R (c, d)) = mkRat (a * d) (b *


type SM<'a> = SM of (rat -> ('a * rat) option)
let ret x = SM (fun st -> Some (x, st))
let bind (SM m) f =
    SM (fun st ->
        match m st with
        | None -> None
        | Some (x, st') ->
            let (SM g) = f x
```

```
                 g st')

    let (>>=) m f = bind m f
    let (>>>=) m n = m >>= (fun () -> n)
    let evalSM (SM f) s = f s

    let smPlus r1  = (SM (fun r2 -> plus r1 r2 |> Opti
    let smMinus r1 = (SM (fun r2 -> minus r2 r1 |> Opt

    let smMult r1  = (SM (fun r2 -> mult r1 r2 |> Opti

    let smDiv r1   = (SM (fun r2 -> div r2 r1 |> Optic

    let rec calculate =
        function
        | []           -> ret ()
        | (r, f) :: xs -> f r >>>= calculate xs

    let calculate2 xs = List.fold (fun acc (r, f) -> a
       (*
    let calculate2 =
        let rec aux acc =
            function
            | []           -> acc
            | (r, f) :: xs -> aux (f r >>>= acc) xs

        function
        | [] -> ret ()
        | (r, f)::xs -> aux (f r) xs
        *)
    type StateBuilder() =

        member this.Bind(x, f)    = bind x f
        member this.Zero ()       = ret ()
        member this.Return(x)     = ret x
        member this.ReturnFrom(x) = x
```

```
        member this.Combine(a, b) = a >>= (fun _ -> b)

    let state = new StateBuilder()

    let rec calculate3 xs =
        state {
            match xs with
            | [] -> return ()
            | (r, f) :: xs ->
                do! f r
                return! calculate3 xs
        }
```

Exam 2021-06-03

https://itu.codejudge.net/func22/collection/3629/view

Question 1

Vores svar

```
type direction = North | East | South | West
type coord = C of int * int

// Question 1.1
let move dist dir (C(x,y))=
    match dir with
    | North -> C (x, y - dist)
    | East -> C (x + dist, y)
    | South -> C (x, y + dist)
    | West -> C (x - dist, y)

let turnRight dir =
    match dir with
    | North -> East
    | East -> South
    | South -> West
    | West -> North
```

```
let turnLeft dir =
    match dir with
    | North -> West
    | East -> North
    | South -> East
    | West -> South

// Question 1.2
type position = P of (coord * direction)
type move = TurnLeft | TurnRight | Forward of int

let step (P(C(x,y), (dir:direction))) (m: move) =
    match m with
    | TurnLeft -> P (C(x,y), turnLeft dir)
    | TurnRight -> P (C(x,y), turnRight dir)
    | Forward dist -> P (move dist dir (C(x,y)), dir)

// Question 1.3
// recursion
let rec walk (P(C(x,y), dir)) (ms: move list) =
    match ms with
    | [] -> P(C(x,y), dir)
    | m::ms -> walk (step (P(C(x,y), dir)) m) ms

// higher order functions
let walk2 (P(C(x,y), dir)) (ms: move list) =
    List.fold (fun acc elem -> step acc elem) (P((C(x,

// Question 1.4
let rec path (P(C(x,y), dir)) (ms: move list) : coord
    match ms with
    | [] -> [C(x,y)]
    | m::ms ->
        match m with
        | Forward _ ->
```

```
            C(x,y)::path (step (P(C(x,y), dir)) m) ms
        | _ -> path (step (P(C(x,y), dir)) m) ms

// Question 1.5
let path2 (P(C(x,y), dir)) (ms: move list) =
    let rec aux (P(C(x,y), dir)) ms acc =
        match ms with
        | [] -> List.rev (C(x,y)::acc)
        | m::ms ->
            match m with
            | Forward _ ->
                aux (step (P(C(x,y), dir)) m) ms (C(x,
            | _ -> aux (step (P(C(x,y), dir)) m) ms ac
    aux (P(C(x,y), dir)) ms []

// Question 1.6
// Explain why path is not recursive, evaluate a funct
// chapter 1.4 in HR.

// tail recursion using continuation
let path3 (P (startCoord, startDir)) moves =
   let rec aux f (P (coord,dir)) =
       function
       | [] -> f []
       | move::moves' ->
           match move with
           | TurnLeft | TurnRight -> (aux f (step (P 
           | Forward dist ->
               let (P (coord',dir')) = step (P (coord,
               aux (fun r -> f (coord'::r)) (P (coord
   aux (fun r -> (startCoord::r)) (P (startCoord, sta
```

▼ Jespers svar

```
module Question1
(* 1: Dungeon crawler *)
```

```
(* Question 1.1 *)

    type direction = North | East | South | West
    type coord     = C of int * int

    let move dist dir (C (x, y)) =
        match dir with
        | North -> C (x, y - dist)
        | East -> C (x + dist, y)
        | South -> C (x, y + dist)
        | West -> C (x - dist, y)

    let turnRight =
        function
        | North -> East
        | East -> South
        | South -> West
        | West -> North

    let turnLeft =
        function
        | North -> West
        | West -> South
        | South -> East
        | East -> North

(* Question 1.2 *)

    type position = P of (coord * direction)
    type move     = TurnLeft | TurnRight | Forward of

    let step (P (c, dir)) =
        function
        | TurnLeft -> P (c, turnLeft dir)
        | TurnRight -> P (c, turnRight dir)
```

```
        | Forward dist -> P (move dist dir c, dir)

(* Question 1.3 *)


    let rec walk p =
        function
        | [] -> p
        | x :: xs -> walk (step p x) xs

    let rec walk2 = List.fold step

(* Question 1.4 *)

    let rec path (P (c, _) as p) =
        function
        | [] -> [c]
        | (Forward _ as x) :: xs -> c::(path (step p x
        | x :: xs                -> path (step p x) xs

(* Question 1.5 *)

    let path2 p ps =
        let rec aux (P (c, _) as p) acc =
            function
            | [] -> List.rev (c::acc)
            | (Forward _ as x) :: xs ->  aux (step p x
            | x :: xs -> aux (step p x) acc xs

        aux p [] ps

(* Question 1.6 *)

(* Q: Your solution for `path` is not tail recursive.
      argument you should evaluate a function call of
      what is done in Chapter 1.4 of HR, and reason ab
```

```
    You need to make clear what aspects of the evalu
    function is not tail recursive. Keep in mind tha
    chain must evaluate to the same value
    (```(5 + 4) * 3 --> 9 * 3 --> 27```, for instanc

A:

  Consider the following derivation
  This is much longer than strictly required (two f
  but it shows a more complete derivation.

  path (P (C (0, 0), North))
       [Forward 5; TurnRight; Forward 5; TurnRight;
        Forward 5; TurnRight; Forward 5] -->

     C (0, 0)::path (step (P (C (0, 0), North)) (Fo
                     [TurnRight; Forward 5; TurnRig

     C (0, 0) :: path (P (C (0, -5), North))
                          [TurnRight; Forward 5; Tu

     C (0, 0) :: path (step (P (C (0, -5), North))
                             [Forward 5; TurnRight;

     C (0, 0) :: path (P (C (0, -5), East))
                          [Forward 5; TurnRight; Fo

     C (0, 0) :: C (0, -5) :: path (step (P (C (0,
                                   [TurnRight; Forw

     C (0, 0) :: C (0, -5) :: path (P (C (5, -5), E
                                   [TurnRight; Forw

    C (0, 0) :: C (0, -5) :: path (step (P (C (5, -
                                  [Forward 5; TurnR
```

```
        C (0, 0) :: C (0, -5) :: path (P (C (5, -5), So
                                       [Forward 5; TurnR

        C (0, 0) :: C (0, -5) :: C (5, -5) :: path (ste
                                                    [Tur

        C (0, 0) :: C (0, -5) :: C (5, -5) :: path (P (
                                                    [Tur

        C (0, 0) :: C (0, -5) :: C (5, -5) :: path (ste
                                                    [For

        C (0, 0) :: C (0, -5) :: C (5, -5) :: path (ste

        C (0, 0) :: C (0, -5) :: C (5, -5) :: C (5, 0)

        C (0, 0) :: C (0, -5) :: C (5, -5) :: C (5, 0)

        [C (0, 0); C (0, -5); C (5, -5); C (5, 0); C ((

        The reason that this function is not tail recur
        cannot be resolved until the final recursive ca
        cons-operations of the coordinate list that kee
        cannot be resolved into a single list until the

*)
    let path3 p ps =
        let rec aux (P (c, _) as p) cont =
            function
            | []                       -> cont [c]
            | (Forward _ as x) :: xs -> aux (step p x)
            | x :: xs                  -> aux (step p x)

        aux p id ps
```

▼ Question 2

▼ Vores svar

```
// Question 2
// Consider and run the following three functions
let foo f =
    let mutable m = Map.empty
    let aux x =
        match Map.tryFind x m with
        | Some y when Map.containsKey x m -> y
        | None ->
            m <- Map.add x (f x) m; f x
    aux

let rec bar x =
    match x with
    | 0 -> 0
    | 1 -> 1
    | y -> baz (y - 1) + baz (y - 2)
and baz = foo bar

// Question 2.1
// What are the types of functions foo , bar , and baz

// foo : ('a -> 'b) -> ('a -> 'b)
// bar : recursive int -> int
// baz : partial function int -> int

// What do functions foo and baz do (skip bar )? Focus

// foo takes a function as a parameter and returns a f
// The function it returns, takes a key as a parameter
// If it does not exists, it saves the result of the c

// bar serves as a recursive function. It takes an int
// baz takes an integer as a parameter and returns an
```

```
// All functions combined serves as the fibonacci sequ

// The function foo uses a mutable variable.
// What function does it serve (why is it there)?

// So that the map can be changed dynamically by the f

// What would happen if you removed the mutable keywor
// Would the function foo still work? If yes, why; if

// No it would not, since the '<-' operator is only al

// What would be appropriate names for functions foo ,

// foo : mapLookup
// bar : aux
// baz : fibonacci

// Question 2.2
// The code includes the keyword and .
// What function does this keyword serve in general (w

// mutual recursion, so that the function can be calle

// What would happen if you removed it from this parti
// the line and baz = foo bar to let baz = foo bar )?

// The bar function would break, as it does not know o

// Question 2.3
// The function foo generates a warning during compila
// expression.
// Why does this happen, and where?

// In the pattern matching of the Map.tryFind. The pat
```

```
// Since we have already checked if the map contains t
// 'Some y' instead of 'Some y when ...' then the patt

// For these particular three functions will this inco
// execution of baz ? If yes, why; if no, why not.

// No, it should not, since the check is redundant. If

// The function foo has two redundant computations and
// two computations and why are they redundant?

// 'Map.containsKey x m' is redundant, since we alread
// '(f x)' and 'f x' it the same computation, this cou

// Write a function foo2 that does exactly the same th
// and is where these two redundant computations have

let foo2 f =
    let mutable m = Map.empty
    let aux x =
        match Map.tryFind x m with
        | Some y -> y
        | None ->
            let r = f x
            m <- Map.add x r m; r
    aux

// Question 2.4
let rec barbaz x =
    let baz = foo barbaz
    match x with
    | 0 -> 0
    | 1 -> 1
    | y -> baz (y - 1) + baz (y - 2)

// baz is slower, I do not know why.
```

```
// Question 2.5
// Write an infinite sequence bazSeq : int seq such th
// second to baz 1 , the third to baz 2 and so on.
// For full credit it must be close to instantanous to
// resulting integers overflow.

// figure out this shit
```

▼ **Jespers svar**

```
module Question2

(* 2: Code Comprehension *)
    let foo f =
        let mutable m = Map.empty
        let aux x =
            match Map.tryFind x m with
            | Some y when Map.containsKey x m -> y
            | None   ->
            m <- Map.add x (f x) m; f x

        aux

    let rec bar x =
      match x with
      | 0 -> 0
      | 1 -> 1
      | y -> baz (y - 1) + baz (y - 2)

    and baz = foo bar

(* Question 2.1 *)

    (*
```

```
Q: What are the types of functions foo, bar, and b

A:
    foo has type ('a -> 'b) -> 'a -> 'b when 'a :
    bar has type int -> int
    baz has type int -> int


Q: What do functions foo and baz do (skip bar)?
   Focus on what they do rather than how they do i

A:
    foo takes a function f and returns a function

    baz takes an integer n and returns the nth fib

    A lot of people wrote something about the muta
    as that is about how the function does somethi
    You could write something like this:

    foo also keeps an internal cache of input-outp
    once for any specific input.

The function foo uses a mutable variable.

Q: What function does it serve (why is it there)?

A: The mutable keyword is used to keep a mutable m

Q: What would happen if you removed the mutable ke
   let mutable m = Map.empty? Would the function f
   If yes, why; if no, why not?

A: No, it would not even compile.

   The <- operation in
```

```
        m <- Map.add x (f x) m
       only works on mutable variables and the progran


    Q: What would be appropriate names for functions
       foo, bar, and baz?

    A:
        foo could be called memoize (it does memoizati
        bar could be called fibAux
        baz could be called fib

    *)

    (*

    A very common problem for this assignment was the
    The mutable state is squarely in how something is
    baz function in particular had a lot of creative s
    All it does is compute fibonacci numbers. If I gav
    from it. Your safest bet would most likely be that

    Keep this in mind. The distinction is important.

    *)


(* Question 2.2 *)


    (*
    The code includes the keyword "and".


    Q: What function does this keyword serve in genera
       (why would you use "and" when writing any progr
```

```
    A: and is typically used to create mutually recurs
       Normally, in F#, functions can only call functi
       but when two functions are dependent on each ot
       has to be declared out of scope of the other. T
       several function be in scope of each other.


    Q: What would happen if you removed it from this p
       replaced it with a standard "let"
       (change the line "and baz = foo bar" to "let ba

    A: The program would not compile as baz would no l
       be able to call baz.


    *)

(* Question 2.3 *)

    (*
    The function foo generates a warning during compil
    "Warning: Incomplete pattern matches on this expre

    Q: Why does this happen, and where?

    A: It happens in the Some case of the pattern matc
            match Map.tryFind x m with
                    | Some y when Map.containsKey x m
                    | None   ->
                    m <- Map.add x (f x) m; f x

       as it contains a guard (Map.containsKey x m) bu

    Q: For these particular three functions will this
       ever cause problems for any possible execution
```

```
       if no, why not.

    A: No, it will not cause a problem as we match on
       reach the Some case if the key x was not in the

    Q: The function foo has two redundant computations
       efficient as it could be. What are these two co
       are they redundant?

    A:
        The first redundant computation is Map.contair
        because otherwise the match of Map.tryFind x r

        The second redundant computation is that we co
        m <- Map.add x (f x) m; f x

    *)

    let foo2 f =
        let mutable m = Map.empty
        let aux x =
            match Map.tryFind x m with
            | Some y -> y
            | None   ->
                let y = f x
                m <- Map.add x y m; y

        aux

(* Question 2.4 *)

    let rec barbaz x =
        let baz = foo barbaz
        match x with
        | 0 -> 0
        | 1 -> 1
```

```
        | y -> baz (y - 1) + baz (y - 2)

    (*

    Q: Without explicitly timing the execution times,
       times of baz and barbaz. One is slower than the
       Why? You do not have to give exact times, just
       slower and explain why.

    A: barbaz is slower as it initializes foo from scr
       What this means is that the mutable map is rese
       in effect negates its effect entirely - previou
       are no longer stored in this map but have to be

    *)
(* Question 2.5 *)

    let bazSeq = Seq.initInfinite baz

    (* Note that this only works because of the cachin
       for these types of problems where an element of
```

▼ Question 3

▼ Phillips løsning

Philips løsning: (3.1&3.2&3.3)

```
type element = (int * int) list

let charToInt c = (int c - int '0')
let elToString (element:element) =
    let rec addToStringXTimes (s:string) (acc:string)
        function
        | 0 -> ""
        | 1 -> acc+s
        | x -> addToStringXTimes s (acc+s) (x-1)
```

```
    List.fold (fun acc ((elId, elAmount):int*int) ->
                addToStringXTimes (string elId) acc el
              ) "" element

let elFromString (stringEl:string) =
    List.fold (
        fun acc (char:char) ->
            let intChar = (charToInt char)
            match acc with
            | [] -> [(intChar, 1)]
            | _ ->
                match acc.Head with
                | idHead, amountHead when intChar = id
                | _ -> (intChar, 1)::acc
        ) [] (List.ofSeq stringEl)
    |> List.rev


let nextElement (element:element) =
    List.fold
        (fun acc (id, amount) ->
                    acc@[(amount,1);(id,1)]
        ) [] (elFromString <| elToString element)
```

▼ Lasses løsning

```
//Question 3

    // Question 3.1
    type element = char list

    let elToString s = List.fold (fun acc elem -> acc
```

```
    let elFromString (str: string) = List.ofSeq str

    let countElements elem =
        let startElement = List.item 0 elem
        let rec aux lst count restOfList =
            match lst with
            | [] -> ((count, startElement), restOfList
            | x :: xs when x = startElement -> aux xs
            | _ :: _ -> ((count, startElement), restOf
        aux elem 0 []

    let nextElement elem =
        let rec aux lst result =
            match lst with
            | [] -> elFromString result
            | xs ->
                let ((count, element), restOfList) = c
                aux restOfList (result + string count
        aux elem ""

// Question 3.4
    let elSeq elem =
        Seq.unfold (fun state -> Some (state, nextElem

    let rec elSeq2 elem =
        seq {yield elem; yield! elSeq2 (nextElement el


// Question 3.5
    open JParsec.TextParser
    open JParsec

    let elParse = many digit .>> pchar '\n' |>> (fun s
//    let elParseAlt = many (digit |>> (int >> (fun x
```

```
    let elFromString2 str =
        run elParse (str + "\n") |> getSuccess
```

Question 4

Vores svar

```
// Question 4.1

    type 'a ring = 'a list * 'a list

// Question 4.2

    let length (a: 'a ring) =
        let (lst1, lst2) = a
        List.length lst1 + List.length lst2

    let ringFromList (lst: 'a list) =
        (List.empty<'a>,  lst)

    let ringToList (ring: 'a ring) =
        let (lst1, lst2) = ring
        lst2 @ List.rev lst1

// Question 4.3

    let empty = (List.empty<'a>, List.empty<'a>)

    let push x (ring: 'a ring) =
        let (lst1, lst2) = ring
        match lst1, lst2 with
        | lst1, [] -> (lst1 @ [x], lst2)
        | lst1, lst2 -> (lst1, x :: lst2)

    let peek (ring: 'a ring) =
        let (lst1, lst2) = ring
        match lst1, lst2 with
```

```
        | [], [] -> None
        | lst1, [] -> Some (List.head (List.rev lst1))
        | _, lst2 when not (List.isEmpty lst2 )-> Some

    let pop (ring: 'a ring) =
        let (lst1, lst2) = ring
        match lst1, lst2 with
        | [], [] -> None
        | lst1, [] -> Some (List.empty<'a>, List.tail
        | lst1, lst2 -> Some (lst1, (List.tail lst2))

    let cw (ring: 'a ring) =
        let (a, b) = ring
        match a, b with
        | [], [] -> List.empty<'a>, List.empty<'a>
        | [], b ->
            let b' = List.rev b
            (List.tail b', [List.head b'])
        | a, b ->
            let x = List.head a
            (List.tail a, x :: b)

    let ccw (ring: 'a ring) =
        let (a, b) = ring
        match a, b with
        | [], [] -> List.empty<'a>, List.empty<'a>
        | a, [] ->
            let a' = List.rev a
            ([List.head a'], List.tail a')
        | a, b ->
            let x = List.head b
            (x :: a, List.tail b)

// Question 4.4
    type StateMonad<'a, 'b> = SM of ('b ring -> ('a *
    let ret x = SM (fun st -> Some (x, st))
```

```
    let bind (SM m) f =
        SM (fun st ->
        match m st with
        | None -> None
        | Some (x, st') ->
            let (SM g) = f x
            g st')

    let (>>=) m f = bind m f
    let (>>>=) m n = m >>= (fun () -> n)
    let evalSM (SM f) s = f s

    let smLength = SM (fun state -> Some (length state
    let smPush x = SM (fun state -> Some ((), push x s
    let smPop = SM (fun state ->
        match peek state with
        | None -> None
        | Some r -> Some (r, pop state |> Option.get)
    let smCW = SM (fun state -> Some ((), cw state))
    let smCCW = SM (fun state -> Some((), ccw state))

// Question 4.5

    type StateBuilder() =
        member this.Bind(x, f) = bind x f
        member this.Zero () = ret ()
        member this.Return(x) = ret x
        member this.ReturnFrom(x) = x
        member this.Combine(a, b) = a >>= (fun _ -> b
    let state = new StateBuilder()

    let ringStep =
        state {
            let! l = smLength
            if l > 1
            then
```

```
                let! x = smPop
                let! y = smPop
                if (x+y) % 2 = 1
                then
                    do! smPush y
                    do! smPush x
                    do! smCCW
        }

    let rec iterRemoveSumEven (x: uint32) =
        state {
            if x > 0u
            then
                do! ringStep
                do! iterRemoveSumEven (x-1u)
        }
```

▼ Jespers svar

```
module Question4

(* 4: Rings *)

(* Question 4.1 *)

    type 'a ring = Ring of 'a list * 'a list

(* Question 4.2 *)

    let length (Ring(xs, ys)) = List.length xs + List
    let ringFromList lst = Ring([], lst)
    let ringToList (Ring(xs, ys)) = ys @ List.rev xs

(* Question 4.3 *)
```

```
    let empty = Ring([], [])
    let push x (Ring(xs, ys)) = Ring(xs, x :: ys)
    let peek =
        function
        | Ring([], [])    -> None
        | Ring(xs, [])    -> Some (List.head (List.rev
        | Ring(_, y :: _) -> Some y
    let pop =
        function
        | Ring([], [])      -> None
        | Ring(xs, [])      -> Some (Ring([], List.ta
        | Ring(xs, _ :: ys) -> Some (Ring(xs, ys))
    let cw =
        function
        | Ring([], [])    -> Ring([], [])
        | Ring([], ys) ->
            let ys' = List.rev ys
            Ring(List.tail ys', [List.head ys'])
        | Ring(x::xs, ys) -> Ring(xs, x::ys)

    let ccw =
        function
        | Ring([], [])    -> Ring([], [])
        | Ring(xs, []) ->
            let xs' = List.rev xs
            Ring([List.head xs'], List.tail xs')
        | Ring(xs, y::ys) -> Ring(y::xs, ys)
(* Question 4.4 *)

    type StateMonad<'a, 'b> = SM of ('b ring -> ('a *
    let ret x = SM (fun st -> Some (x, st))
    let bind (SM m) f =
        SM (fun st ->
            match m st with
            | None -> None
```

```
            | Some (x, st') ->
                let (SM g) = f x
                g st')

    let (>>=) m f = bind m f
    let (>>>=) m n = m >>= (fun () -> n)
    let evalSM (SM f) s = f s


    let isEmpty (Ring(xs, ys)) = List.isEmpty xs && Li

    let smLength = SM(fun s -> Some (length s, s))
    let smPush x = SM (fun s -> Some((), push x s))
    let smPop = SM (fun s -> if isEmpty s then None el
    let smCW = SM (fun s -> Some ((), cw s))
    let smCCW = SM (fun s -> Some ((), ccw s))

(* Question 4.4 *)

    let ringStep : StateMonad<unit, int> =
        smLength >>= fun l ->
        if l > 1 then
            smPop >>= fun x ->
            smPop >>= fun y ->
            if (x + y) % 2 = 0 then
                ret ()
            else
                smPush y >>>=
                smPush x >>>=
                smCCW
        else
            ret ()

    let rec iterRemoveSumEven =
        function
        | 0u -> ret ()
```

```
        | x  -> ringStep >>>= iterRemoveSumEven (x - 1

    (* You may solve this exercise either using monadi
        using computational expressions. *)

    type StateBuilder() =

        member this.Bind(x, f)    = bind x f
        member this.Zero ()       = ret ()
        member this.Return(x)     = ret x
        member this.ReturnFrom(x) = x
        member this.Combine(a, b) = a >>= (fun _ -> b)

    let state = new StateBuilder()

    let ringStep2 () : StateMonad<unit, int> =
        state {
            let! l = smLength
            if l > 0 then
                let! x = smPop
                let! y = smPop
                if x % 2 = 0 then
                    return ()
                else
                    do! smPush y
                    do! smPush x
                    do! smCCW
        }

    let rec iterRemoveSumEven2 x =
        state {
            if x > 0 then
                do! ringStep2 ()
                do! iterRemoveSumEven2 (x - 1)
        }
```

▼ Thor og Co.

```
module FPExam2020.Exam2021

    open JParsec
    open JParsec.TextParser
    open Microsoft.FSharp.Collections
    //1. Dungeon Crawler

    type direction = North | East | South | West
    type coord = C of int * int

    //1.1
    let move (dist: int) (dir: direction) (C(x, y)) =
        match dir with
        | North -> C(x, y - dist)
        | South -> C(x, y + dist)
        | West -> C(x - dist, y)
        | East -> C(x + dist, y)

    let turnRight (dir: direction) =
        match dir with
        | North -> East
        | East -> South
        | South -> West
        | West -> North

    let turnLeft (dir: direction) =
        match dir with
        | North -> West
        | East -> North
        | South -> East
        | West -> South

    move 10 North (C (0, 0))
```

```
    //turnRight North
    //turnLeft North

    //1.2
    type position = P of (coord * direction)
    type move = TurnLeft | TurnRight | Forward of int

    let step (P(crd, dir)) (m: move) =
        match m with
        | TurnLeft -> P(crd, turnLeft dir)
        | TurnRight -> P(crd, turnRight dir)
        | Forward x -> P((move x dir crd), dir)

    //step (P (C (0, 0), North)) TurnRight
    //step (P (C (0, 0), North)) TurnLeft
    //step (P (C (0, 0), North)) (Forward 10)

    //1.3
    let rec walk (p: position) (ms: move list) =
        match ms with
        | head :: tail -> walk (step p head) tail
        | _ -> p

    //walk (P (C (0, 0), North)) [TurnRight; Forward 10;

    let walk2 (p: position) (ms: move list) =
        List.fold(fun acc move ->
                    step acc move
                ) p ms

    //walk2 (P (C (0, 0), North)) [TurnRight; Forward 10

    //1.4
    let rec path (P(crd, dir)) (ms: move list) =

        let rec aux (P(crd, dir)) (ms: move list) =
```

```
            match ms with
            | [] -> []
            | head :: tail ->
                let (P(crd2, dir2)) = step (P(crd, dir))
                match head with
                | TurnLeft -> aux (P(crd2, dir2)) tail
                | TurnRight -> aux (P(crd2, dir2)) tail
                | Forward _ -> crd2 :: aux (P(crd2, dir2

        crd :: aux (P(crd, dir)) ms

    path (P (C (0, 0), North)) [TurnRight; Forward 10; T

    path (P (C (0, 0), North))
     [Forward 5; TurnRight; Forward 5; TurnRight;
     Forward 5; TurnRight; Forward 5];

    //1.5
    let rec path2 (P(crd, dir)) (ms: move list) =
        let rec aux (P(crd, dir)) (ms: move list) (acc:
            match ms with
            | [] -> acc
            | head :: tail ->
                let (P(crd2, dir2)) = step (P(crd, dir))
                match head with
                | TurnLeft -> aux (P(crd2, dir2)) tail a
                | TurnRight -> aux (P(crd2, dir2)) tail
                | Forward _ -> aux (P(crd2, dir2)) tail
        aux (P(crd, dir)) ms [crd]

    //1.6
    let rec path3 (P(crd, dir)) (ms: move list) =

        let rec aux (P(crd, dir)) (ms: move list) cont =
            match ms with
            | [] -> cont []
```

```
            | head :: tail ->
                let (P(crd2, dir2)) = step (P(crd, dir))
                match head with
                | TurnLeft -> aux (P(crd2, dir2)) tail c
                | TurnRight -> aux (P(crd2, dir2)) tail
                | Forward _ -> aux (P(crd2, dir2)) tail
        aux (P(crd, dir)) ms (fun crdLst -> [crd]@crdLst

path3 (P (C (0, 0), North))
     [Forward 5; TurnRight; Forward 5; TurnRight;
     Forward 5; TurnRight; Forward 5]

//2.0
let foo f =
    let mutable m = Map.empty
    let aux x =
        match Map.tryFind x m with
        | Some y when Map.containsKey x m -> y
        | None -> m <- Map.add x (f x) m; f x
    aux

let foo2 f =
    let mutable m = Map.empty
    let aux x =
        match Map.tryFind x m with
        | Some y -> y
        | None ->
            let result = f x
            m <- Map.add x (result) m; result
    aux

let fooImmutable f =
    let m = Map.empty
    let aux x =
        match Map.tryFind x m with
        | Some y when Map.containsKey x m -> y
```

```
            | None -> Map.add x (f x) m; f x
        aux

    let rec bar x =
        match x with
        | 0 -> 0
        | 1 -> 1
        | y -> baz (y - 1) + baz (y - 2)

    and baz = foo bar
    and baz2 = foo2 bar
    and bazImmutable = fooImmutable bar

    //2.1

    //Foo is not recursive
    //Bar is recursive but not tail-recursive
    //The methods are used to get fibonaci numbers
    //You can for example use baz 2, to the fib number o

    //Mutable is useful because it saves the values in t
    //Without the map being mutable, it would take much 

    //If you removed the mutable keyword it would not wo
    //But if you also removed the "<-" assignment, the f
    //But the new map would never be used.
    //We can still however get the correct result with b
    //But without it being mutable we would have to comp

    //foo: setupMapThenGetValueOrAddIfNotExisting
    //bar: aux
    //baz: calculateFib
    //

    //2.2
    //and keyword: Used in mutually recursive bindings a
```

```
//in property declarations, and with multiple constr

//It would not work if we used "let" because baz nee

//2.3

//The function foo generates a warning during compil
//Warning: Incomplete pattern matches on this expres

//This issue happens because of the check with 'when
//This would never cause an issue because we already
//tryFind will only return Some if there's an value,
//need to check again. But the compiler might think
//that the pattern matching is incomplete.

//Redundant computations:
//Map.containsKey could be removed because you alrea
//Removing this also removes the "incomplete" warnin
//
//f x is calculated twice and it can be replaced by
//

//2.4
let rec barbaz x =
    let baz = foo barbaz
    match x with
    | 0 -> 0
    | 1 -> 1
    | y -> baz (y - 1) + baz (y - 2)

//barbaz is slower.
//Everytime you run barbaz you define 'let baz = foo
//Which means that 'foo' will run many times.

//2.5
//
```

```
let bazSeq: int seq = Seq.initInfinite(baz2)

Seq.item 10 bazSeq

//3.1

type element = int list

//3.2
let elToString (e: element) =
    List.fold(fun acc valu ->
                    acc + string valu
                ) "" e

elToString [1;2;5;1;3;3]


let inline charToInt c = int c - int '0'

let elFromStringAlt (str: string) =
    List.fold(fun acc valu ->
                    acc @ [charToInt valu]
                ) [] (List.ofSeq(str))
let elFromStringAlt2 (str: string) =
    List.foldBack(fun valu acc ->
                    charToInt valu :: acc
                ) (List.ofSeq(str)) []

let elFromString (str: string) =
    let lst = List.ofSeq(str)

    let rec aux (chrLst: char list) =
        match chrLst with
        | head :: tail -> charToInt head :: aux tail
        | _ -> []
```

```
    aux lst

elFromString "5461254"
elFromStringAlt "5461254"
elToString (elFromString "1113221")

//3.3

let countList (count: int) =
    if count = 10 then
        [1;0]
    else
        [count]

let nextElement (e1: element): element =
    let rec aux (e1: element) (previousNumber: int)
        match e1 with
        | head :: tail ->
            if head = previousNumber || previousNumb
                aux tail head (count + 1) acc
            else
                if List.length tail >= (List.length
                    aux tail head 1 (acc @ countList
                else
                    aux tail head 0 (acc @ countList
        | _ -> acc @ countList count @ [previousNumb

    aux e1 -1 0 List<int>.Empty

//1
//11
//21
//1211
//111221
```

```
"1" |> elFromString |> nextElement |> elToString
"1" |> elFromString |> nextElement |> nextElement |>

"1" |> elFromString |> nextElement |> nextElement |>
    nextElement |> nextElement |> nextElement |> elT

"1111111111" |> elFromString |> nextElement



//3.4
//let elSeq (e1: element) : element seq = Seq.initIn
let elSeq (e1: element) =
    Seq.unfold (fun state -> Some (state, nextElemen

let elSeq2 (e1: element) = seq { for i in 1 .. 10 do

//let elSeq (e1: element) : element seq = Seq.initIn
//Seq.initInfinite requires an (int -> 'T)
//Our nextElement is of type element -> element.
//The int is an index, but our list of elements does
//The sequence can vary depending on the starting el

//3.5

let whitespaceChar = satisfy System.Char.IsWhiteSpac
let pletter        = satisfy System.Char.IsLetter <?
let palphanumeric  = satisfy System.Char.IsLetterOrD
let pdigit  = satisfy System.Char.IsDigit <?> "digit

let spaces         = many (whitespaceChar) <?> "spac
let spaces1        = many1 (whitespaceChar) <?> "spa
let newLine        = pstring "\\n"

let (.>*>.) p1 p2 = p1 .>> spaces .>>. p2
let (.>*>) p1 p2  = p1 .>> spaces .>> p2
let (>*>.) p1 p2  = p1 >>. spaces >>. p2
```

```
let myParse = spaces >*>. many pint32 .>*> spaces

let charListToIntList (chrLst: char list) =
    List.foldBack(fun valu acc ->
                    charToInt valu :: acc
               ) chrLst []
//101
let elParse: Parser<element> = spaces >*>. many digi
                                          |>> (fun a -> charLi

let elFromString2 (str: string): element =
    run elParse str |> getSuccess

//let pid = pchar '' <|> pletter .>>. many (palphanu

//4.1
type ring<'a> = 'a list * 'a list

//4.2
let length (aLst, bLst) =
    List.length aLst + List.length bLst

let lengthAlt (r: 'a ring) =
    let (aLst, bLst) = r
    List.length aLst + List.length bLst

let ringFromList (lst: 'a list) =
    (List<'a>.Empty, lst)

let ringToList ((aLst, bLst): 'a ring) =
    bLst @ List.rev aLst

//ringToList (ringFromList [1;2;3;4;5])
//length (ringFromList [1;2;3;4;5])
//ringToList (ringFromList [1;2;3;4;5])
```

```
//4.3
let empty<'a> : 'a ring = (List.Empty, List.Empty)
let push (value: 'a) (aLst, bLst) =
    match bLst with
    | [] -> (aLst @ [value], bLst)
    | _ -> (aLst, value :: bLst)

//[1;2;3;4;5] |> ringFromList |> push 6 |> ringToLis

let peek (aLst, bLst) =
    match bLst with
    | [] ->
        match List.rev aLst with
        | [] -> None
        | head :: _ -> Some(head)
    | head :: _ -> Some(head)

//[1;2;3;4;5] |> ringFromList |> peek

let pop ((aLst, bLst): 'a ring) =
    match bLst with
    | [] ->
        match List.rev aLst with
        | [] -> None
        | _ :: tail -> Some ((tail, bLst): 'a ring)
    | _ :: tail -> Some ((aLst, tail): 'a ring)

//[1;2;3;4;5] |> ringFromList |> pop |> ringToList /

let cw ((aLst: 'a list, bLst: 'a list)) =
    match aLst, bLst with
    | [], [] -> ([], [])
    | [], bList2 ->
        let rev = List.rev bList2
        let x = rev.Head
```

```
            let c = rev.Tail
            (c, [x])

        | aList2, bList2 ->
            let x = aList2.Head
            let c = aList2.Tail
            (c, (x :: bList2))

    //[1;2;3;4;5] |> ringFromList |> cw |> cw |> ringToL

    let ccw ((aLst: 'a list, bLst: 'a list)) =
        match aLst, bLst with
        | [], [] -> ([], [])
        | aList2, [] ->
            let rev = List.rev aList2
            let x = rev.Head
            let c = rev.Tail
            ([x], c)

        | aList2, bList2 ->
            let x = bList2.Head
            let c = bList2.Tail
            ((x :: aList2), c)

    [1;2;3;4;5] |> ringFromList |> ccw |> ccw |> ringToL

    //4.4

    type StateMonad<'a, 'b> = SM of ('b ring -> ('a * 'b
    let ret x = SM (fun st -> Some (x, st))
    let bind (SM m) f =
        SM (fun st ->
            match m st with
            | None -> None
            | Some (x, st') ->
                let (SM g) = f x
```

```
            g st')

let (>>=) m f = bind m f
let (>>>=) m n = m >>= (fun () -> n)
let evalSM (SM f) s = f s

let smLength = SM(fun ring -> Some(length ring, ring
let smPush (x: 'a) = SM(fun ring -> Some((), push x
let smPop = SM(fun ring ->
    match peek ring with
    | Some value ->
        let poppedRing = pop ring |> Option.get
        Some(value, poppedRing)
    | None -> None)

let smCW = SM(fun ring -> Some((), cw ring))
let smCCW = SM(fun ring -> Some((), ccw ring))

[1;2;3;4;5] |> ringFromList |> evalSM smLength |> Op
[1;2;3;4;5] |> ringFromList |> evalSM (smPush 6) |>
[1;2;3;4;5] |> ringFromList |> evalSM smPop |> Optio
([] : int list) |> ringFromList |> evalSM smPop

[1;2;3;4;5] |> ringFromList |> evalSM (smCW >>>= smC
        Option.get |> snd |> ringToList

[1;2;3;4;5] |> ringFromList |> evalSM (smCCW >>>= sm
Option.get |> snd |> ringToList

//4.5
```

Exam 2020-08-17

Question 1

```
(* Question 1.1 *)
let rec insert elem tree =
        match tree with
        | Leaf                                -> Node(L
        | Node (left, y, right) when elem <= y -> Node(i
        | Node (left, y, right)                -> Node(l

(* Question 1.2 *)
    let fromList lst =
        let rec aux acc lst =
            match lst with
            | []            -> acc
            | head :: rest -> aux (insert head acc) rest
        aux Leaf lst

(* Question 1.3 *)
let rec fold (f: ('a -> 'b -> 'a)) (acc: 'a) (tree: 'b b
    match tree with
    | Leaf                  -> acc
    | Node (Leaf, node, Leaf)  -> f acc node
    | Node (left, _, right) -> fold f (fold f acc left)
```

Question 2

Vores svar

```
module Exam
(* If you are importing this into F# interactive then
   the line above and remove the comment for the line

   Do note that the project will not compile if you do
   it does allow you to work in interactive mode and y
   to make the project compile work again.

   Do not remove the line (even though that does work)
   introduce indentation errors in your code that may
```

```
   to switch back to project mode.

   Alternative, keep the line as is, but load ExamInte
   *)
(* module Exam2020_2 = *)

(* 1: Binary search trees *)
    type 'a bintree =
    | Leaf
    | Node of 'a bintree * 'a * 'a bintree

(* Question 1.1 *)
    let rec insert elem tree =
        match tree with
        | Leaf                                  -> Node
        | Node (left, y, right) when elem <= y -> Node
        | Node (left, y, right)                -> Node

(* Question 1.2 *)
    let fromList lst =
        let rec aux acc lst =
            match lst with
            | []          -> acc
            | head :: rest -> aux (insert head acc) re
        aux Leaf lst

(* Question 1.3 *)
    let rec fold (f: ('a -> 'b -> 'a)) (acc: 'a) (tree
        match tree with
        | Leaf                      -> acc
        | Node (left, node, right) ->
            let leftAcc = fold f acc left
            let currentAcc = f leftAcc node
            let rightAcc = fold f currentAcc right
            rightAcc
```

```
    let rec foldBack (f: ('a -> 'b -> 'a)) (acc: 'a) (
        match tree with
        | Leaf                   ->
            acc
        | Node (left, node, right) ->
            let rightAcc = fold f acc right
            let currentAcc = f rightAcc node
            let leftAcc = fold f currentAcc left
            leftAcc

    let inOrder (tree: 'a bintree) = foldBack (fun acc


(* Question 1.4 *)

    (*

    Q: Consider the following map function

    *)

    let rec badMap f =
      function
      | Leaf -> Leaf
      | Node (l, y, r) -> Node (badMap f l, f y, badMa

    (*
    Even though the type of this function is `('a ->
    as we would expect from a map function, this  func
    we want it to do. What is the problem? Provide an

    A: Does not keep order in tree
    *)

    let rec map f tree =
        fold (fun acc elem -> (f elem) :: acc) [] tree
```

```
(* 2: Code Comprehension *)
    let rec foo =
        function
        | [x]                 -> [x]
        | x::y::xs when x > y -> y :: (foo (x::xs))
        | x::xs               -> x :: foo xs

    let rec bar =
        function
        | [x]          -> true
        | x :: y :: xs -> x <= y && bar (y :: xs)

    let rec baz =
        function
        | []              -> []
        | lst when bar lst -> lst
        | lst             -> baz (foo lst)


(* Question 2.1 *)

    (*

    Q: What are the types of functions foo,  bar, and

    A:
        foo: _arg1: 'a list -> 'a list when 'a: compar
        bar: _arg1: 'a list -> bool when 'a: compariso
        baz: _arg1: 'a list -> 'a list when 'a: compar


    Q: What do functions ```bar```, and ```baz``` do
       (not `foo`, we admit that it is a bit contrived
       Focus on what they do rather than how they do i
```

```
    A:
        bar: takes 1 argument, a list, and checks if i
        baz: takes 1 argument, a list, and sorts it.


    Q: What would be appropriate names for functions
       foo, bar, and baz?

    A:
        bar: isSorted
        baz: sort

    *)


(* Question 2.2 *)


    (*
    The functions foo and bar generate a warning durin
    'Warning: Incomplete pattern matches on this expre

    Q: Why does this happen, and where?

    A:
        in the match case, because they dont match on


    Q: For these particular three functions will this
       pattern match ever cause problems for any possi
       If yes, why; if no, why not.

    A: No, because baz matches on empty list. So no er

    *)
```

```
    let rec foo2 =
        function
        | []                   -> []
        | [x]                  -> [x]
        | x::y::xs when x > y -> y :: (foo (x::xs))
        | x::xs                -> x :: foo xs

    let rec bar2 =
        function
        | []          -> true
        | [x]         -> true
        | x :: y :: xs -> x <= y && bar (y :: xs)

    (* Uncomment code to run after you have written fo
    let rec baz2 =
      function
      | lst when bar2 lst -> lst
      | lst               -> baz2 (foo2 lst)


(* Question 2.3 *)

    (* Consider this alternative definition of *)

    let rec foo3 =
      function
      | [x]                  -> [x]
      | x::xs                -> x :: foo3 xs
      | x::y::xs when x > y -> y :: (foo3 (x::xs))

    (*

    Q: Do the functions `foo` and `foo3` produce the s
       If yes, why; if no why not and provide a counte

    A:
```

```
        No because foo3 calls x :: foo3 xs, which mear

        foo3 [1;2;3;2];;
        val it: int list = [1; 2; 3; 2]

        foo [1;2;3;2];;
        val it: int list = [1; 2; 2; 3]

    *)

(* Question 2.4 *)

//    let rec bar2 =
//        function
//        | []           -> true
//        | [x]          -> true
//        | x :: y :: xs -> x <= y && bar (y :: xs)

    let bar3 lst = List.mapi (
        fun index elem ->
            match index with
            | 0 -> true
            | index -> List.item (index-1) lst <= elem
                    ) lst |> List.forall (fun r -> r =

(* Question 2.5 *)

    (*

    Q: The function foo or baz is not tail recursive.

    A:
        foo is not tail recursive, since it calls y :

    *)
```

```
    (* ONLY implement the one that is NOT already tail

//        let rec foo =
//        function
//        | [x]                 -> [x]
//        | x::y::xs when x > y -> y :: (foo (x::xs))
//        | x::xs               -> x :: foo xs


    let fooTail lst =
        let rec aux lst c =
            match lst with
            | [] -> c []
            | [x] -> c [x]
            | x::y::xs when x > y -> aux (x::xs) (fun
            | x :: xs -> aux xs (fun r -> c (x :: r))
        aux lst id

(* 3: Big Integers *)

(* Question 3.1 *)

    type bigInt = int list (* replace unit with the co

    let fromString (nums: string): bigInt =
        (List.foldBack (fun character acc -> int chara

    let toString (x: bigInt) =
        List.foldBack (fun currentInt acc -> string cu

(* Question 3.2 *)

    let add (x: bigInt) (y: bigInt): bigInt =

            let num = (List.length x) - (List.length y
```

```
            let zList = [for i in 1 .. (abs num) -> 0]

            let x' = List.rev x
            let y' = List.rev y

            let rec aux (first: bigInt) (second: bigIn
                match first, second with
                | [], [] when extra = 1 -> [1] @ fromS
                | [], [] -> fromString acc
                | x::xs, y::ys ->
                    if (x + extra)+y >= 10 then
                        aux xs ys (string (((x + extra
                    else
                        aux xs ys (string ((x + extra)

            if num > 0 then
                aux x' (List.rev (zList @ y)) "" 0
            else
                aux (List.rev (zList @ x)) y' "" 0

(* Question 3.3 *)
    let multSingle (a: bigInt) (b: int) =
        if a = [0] || b = 0 then [0]
        else

        let rec aux i acc =
            match i with
            | _ when i = b -> acc
            | i' -> aux (i' + 1) (add a acc)
        aux 0 [0]

(* Question 3.4 *)

    let mult _ = failwith "not implemented"

(* Question 3.5 *)
```

```
    let fact _ = failwith "not implemented"

(* 4: Lazy lists *)

    type 'a llist =
    | Cons of (unit -> ('a * 'a llist))

    let rec llzero = Cons (fun () -> (0, llzero))

(* Question 4.1 *)

    let step (ll: 'a llist) =
        match ll with
        | Cons a ->
            let (hd, tl) = a ()
            (hd, tl)

    let cons (x: 'a) (ll: 'a llist) = Cons (fun () ->

(* Question 4.2 *)

    let rec index (f: int -> 'a) num = Cons (fun () ->

    let init (f: int -> 'a) = index f 0

(* Question 4.3 *)

    let llmap _ = failwith "not implemented"

(* Question 4.4 *)

    let filter _ = failwith "not implemented"

(* Question 4.5 *)
```

```
    let takeFirst _ = failwith "not implemented"

(* Question 4.6 *)

    let unfold _ = failwith "not implemented"

    (* Consider the following two implementations of F

    let fib x =
        let rec aux acc1 acc2 =
            function
            | 0 -> acc1
            | x -> aux acc2 (acc1 + acc2) (x - 1)

        aux 0 1 x

    (* Uncomment after you have implemented init and u

(*
    let fibll1 = init fib
    let fibll2 = unfold (fun (acc1, acc2) -> (acc1, (a
  *)
    (*

    Q: Both fibll1 and fibll2 correctly calculate a la
       Which of these two lazy lists is the most effic

    A: <Your answer goes here>

    *)
```

Jespers svar

```
module Exam2020_2

(* 2: Code Comprehension *)
```

```
    let rec foo =
        function
        | [x]                 -> [x]
        | x::y::xs when x > y -> y :: (foo (x::xs))
        | x::xs               -> x :: foo xs

    let rec bar =
        function
        | [x]          -> true
        | x :: y :: xs -> x <= y && bar (y :: xs)

    let rec baz =
        function
        | []               -> []
        | lst when bar lst -> lst
        | lst              -> baz (foo lst)


(* Question 2.1 *)

    (*

    Q: What are the types of functions foo,  bar, and

    A:

    foo has type 'a list -> 'a list when 'a : comparis
    bar has type 'a list -> bool when 'a : comparison
    baz has type 'a list -> 'a list when 'a : comparis


    Q: What do functions ```bar```, and ```baz``` do
       (not `foo`, we admit that it is a bit contrived
       Focus on what they do rather than how they do i

    A:
```

```
        bar takes a list and returns true if that list
        baz sorts a list in increasing order


    Q: What would be appropriate names for functions
       foo, bar, and baz?

    A: These three functions implement bubble sort. Th
        strictly required to know.

        foo could be called bubble (for bubble sort) o
        bar could be called isOrdered
        baz could be called sort or bubbleSort

    *)


(* Question 2.2 *)


    (*
    The functions foo and bar generate a warning durin
    'Warning: Incomplete pattern matches on this expre

    Q: Why does this happen, and where?

    A: Both foo and bar require that the argument list
       are empty is not covered by the pattern match.
       is not covered by either function.


    Q: For these particular three functions will this
       pattern match ever cause problems for any possi
       If yes, why; if no, why not.

    A: No it will not. The function baz checks if its
```

```
      call either foo or bar with an empty list.

   *)

   let rec foo2 =
       function
       | []                  -> []
       | x::y::xs when x > y -> y :: (foo2 (x::xs))
       | x::xs               -> x :: foo2 xs

   let rec bar2 =
       function
       | []           -> true
       | [x]          -> true
       | x :: y :: xs -> x <= y && bar2 (y :: xs)

   (* Uncomment code to run after you have written fo

   let rec baz2 =
     function
     | lst when bar2 lst -> lst
     | lst               -> baz2 (foo2 lst)


(* Question 2.3 *)

   (* Consider this alternative definition of *)

   let rec foo3 =
     function
     | [x]                 -> [x]
     | x::xs               -> x :: foo3 xs
     | x::y::xs when x > y -> y :: (foo3 (x::xs))

   (*
```

```
    Q: Do the functions `foo` and `foo3` produce the s
       If yes, why; if no why not and provide a counte

    A: No, they do not. By swapping the second and thi
       as matches are always checked in order. The fun

       Counter example

       foo3 [9;1;2;3] = [9;1;2;3] <> [1;2;3;9] = foo

    *)

(* Question 2.4 *)

    let bar3 =
        function
        | [] -> true
        | x :: xs ->
            List.fold (fun (b, x) y -> (b && x < y, y)
            fst

(* Question 2.5 *)

    (*

    Q: The function foo or baz is not tail recursive.

    A: The function foo is not tail recursive as the f

    foo [9;1;2;3] -->
    1 :: (foo (9::[2;3])) -->
    1 :: 2 :: (foo (9::[3])) -->
    1 :: 2 :: 3 :: (foo (9::[])) -->
    1 :: 2 :: 3 :: [9] -->
    [1;2;3;9]
```

```
    The cons-operators in every step build up and a va
    This can eventually lead to a stack overflow for l

    *)

    (* ONLY implement the one that is NOT already tail

    let fooTail lst =
        let rec aux c =
            function
            | [] -> c []
            | x :: y :: xs when x > y ->
                aux (fun result -> c (y :: result)) (x
            | x :: xs ->
                aux (fun result -> c (x :: result)) xs

        aux id lst
```

▼ Question 3

▼ Solution 1 (original)

```
(* 3: Big Integers *)

(* Question 3.1 *)

    type bigInt = int list (* replace unit with the c

    let fromString (nums: string): bigInt =
        (List.foldBack (fun character acc -> int char

    let toString (x: bigInt) =
        List.foldBack (fun currentInt acc -> string c

(* Question 3.2 *)
```

```
let add (x: bigInt) (y: bigInt): bigInt =

        let num = (List.length x) - (List.length
        let zList = [for i in 1 .. (abs num) -> 0

        let x' = List.rev x
        let y' = List.rev y

        let rec aux (first: bigInt) (second: bigI
            match first, second with
            | [], [] when extra = 1 -> [1] @ from
            | [], [] -> fromString acc
            | x::xs, y::ys ->
                if (x + extra)+y >= 10 then
                    aux xs ys (string (((x + extr
                else
                    aux xs ys (string ((x + extra

        if num > 0 then
            aux x' (List.rev (zList @ y)) "" 0
        else
            aux (List.rev (zList @ x)) y' "" 0

    (* Alternate add *)
    let add (x:bigInt) (y:bigInt) =
    let length = (max x.Length y.Length)
    let rec aux index carry (acc:bigInt) =
        if index <= length || carry > 0
        then
            let xValue = if x.Length >= index the
            let yValue = if y.Length >= index the
            let addition = xValue + yValue + carr
            let resultInPosition = addition % 10
            let carryOver = (addition-resultInPos
            aux (index+1) carryOver (resultInPosi
        else acc
```

```
        aux 1 0 []

(* Question 3.3 *)
    let multSingle (a: bigInt) (b: int) =
        if a = [0] || b = 0 then [0]
        else

        let rec aux i acc =
            match i with
            | _ when i = b -> acc
            | i' -> aux (i' + 1) (add a acc)
        aux 0 [0]
```

▼ Solution 2 (Alternative - Philip - Inklusiv async opgaver)

```
(* 3: Big Integers *)

(* Question 3.1 *)

    type bigInt = int list

    let fromString (s:string) : bigInt =
        Seq.map (fun c -> int c - int '0') s |> List.
    let toString (bigInt:bigInt) = List.fold (fun acc

(* Question 3.2 *)

    let add (bigInt1:bigInt) (bigInt2:bigInt) =
        let (longBoi, smallBoi) = if List.length bigI
        let smallLength = List.length smallBoi
        let longLength = List.length longBoi

        let rec aux overflow (i:int) (acc:bigInt) =
            match List.tryItem (longLength - i) longB
            | None -> if overflow > 0 then (overflow:
```

```
            | Some x ->
                let r = match List.tryItem (smallLeng
                        | None ->
                            x+overflow
                        | Some y ->
                            x+y+overflow
                match r with
                | r when r >= 10 -> aux (r/10) (i + 1
                | r-> aux 0 (i + 1) (r::acc)
        aux 0 1 []

    let rec removeHeadZero (bigInt:bigInt) =
        match bigInt with
        | [] -> [0]
        | head::tail when head = 0 -> removeHeadZero
        | _ -> bigInt

    (* Question 3.3 *)
    let multSingle (bigInt:bigInt) (multiplier:int) =
        let mutable i = 0
        List.foldBack (
            fun n acc ->
                let list = [for _ in 1..i -> 0]
                let r =
                    match (n*multiplier) with
                    | r when r >= 10 -> [r/10; r%10]
                    | r -> [r]
                    @ list
                i <- i + 1
                add r acc
        ) bigInt []
        |> removeHeadZero

(* Question 3.4 *)

    let mult (bigInt:bigInt) (multiplier:bigInt) =
```

```
        let mutable i = 0
        List.foldBack (
            fun bigN acc ->
                let r = multSingle bigInt bigN @ [for
                i <- i + 1
                add r acc
        ) multiplier []
        |> removeHeadZero

(* Question 3.5 *)

    let intToBigInt x =  fromString <| string x

    let fact x numThreads : bigInt=
        match x with
        | 0 -> [1]
        | x ->
            let amountPerThead = x/numThreads
            let threads = [for i in 0..numThreads-1 -
            List.map (
                fun thread ->
                    async {
                        return List.fold mult [1] thr
                    }
            ) threads
            |> Async.Parallel
            |> Async.RunSynchronously
            |> Array.fold mult [1]
```

▼ Question 4

```
(* 4: Lazy lists *)

    type 'a llist =
    | Cons of (unit -> ('a * 'a llist))
```

```
    let rec llzero = Cons (fun () -> (0, llzero))

(* Question 4.1 *)

    let step (ll: 'a llist) =
        match ll with
        | Cons a ->
            let (hd, tl) = a ()
            (hd, tl)

    let cons (x: 'a) (ll: 'a llist) = Cons (fun () -> (x

(* Question 4.2 *)

    let init (f: int -> 'a) =
        Cons (fun () -> (f 0, (cons 0 llzero)))

(* Question 4.3 *)

    let llmap _ = failwith "not implemented"

(* Question 4.4 *)

    let filter _ = failwith "not implemented"

(* Question 4.5 *)

    let takeFirst _ = failwith "not implemented"

(* Question 4.6 *)

    let unfold _ = failwith "not implemented"

    (* Consider the following two implementations of Fib

    let fib x =
```

```
        let rec aux acc1 acc2 =
            function
            | 0 -> acc1
            | x -> aux acc2 (acc1 + acc2) (x - 1)

        aux 0 1 x

    (* Uncomment after you have implemented init and unf

(*
    let fibll1 = init fib
    let fibll2 = unfold (fun (acc1, acc2) -> (acc1, (acc
  *)
    (*

    Q: Both fibll1 and fibll2 correctly calculate a lazy
       Which of these two lazy lists is the most efficie

    A: <Your answer goes here>

    *)
```

Exam 2020-05-25

Question 1

lasse svar

```
let rec insert e lst =
        match lst with
        | [] -> [e]
        | x :: xs when e <= x -> e::x::xs
        | x :: xs -> x :: insert e xs

    let rec insertionSort lst =
        match lst with
        | [] -> []
```

```
        | x :: xs -> insert x (insertionSort xs)

(* Question 1.2 *)

    let insertTail e lst =
        let rec aux lst acc =
            match lst with
            | [] -> e::acc
            | x :: xs when e < x -> (List.rev <| e::x
            | x :: xs -> aux xs (x :: acc)
        List.rev <| aux lst []


    let insertionSortTail lst =
        let rec aux lst acc =
            match lst with
            | [] -> acc
            | x :: xs -> aux xs (insertTail x acc)
        aux lst []

(* Question 1.3 *)

    (*
    Q: Why are the higher-order functions from the Lis
    not a good fit to implement insert?

    In the insert function we have two base cases, the
    element fits into a given spot.
    Higher order function does not support this. Highe
    support not itterating through the whole list. We
    returning in the middle of a fold

    *)

    let insertionSort2 lst = List.fold (fun acc elem -
```

```
(* Question 1.4 *)

    let insertBy f e lst =
            let rec aux lst acc =
                match lst with
                | [] -> e::acc
                | x :: xs when f e < f x -> (List.rev
                | x :: xs -> aux xs (x :: acc)
            List.rev <| aux lst []

    let insertionSortBy f lst =
        List.fold (fun acc elem -> insertBy f elem acc
```

▼ Jesper svar

```
module Exam2020Q1

(* 1: Insertion sort *)

(* Question 1.1 *)

    let rec insert x =
        function
        | [] -> [x]
        | y :: ys when y < x -> y :: (insert x ys)
        | y :: ys            -> x :: y :: ys

    let rec insertionSort =
        function
        | [] -> []
        | x :: xs -> insert x (insertionSort xs)

(* Question 1.2 *)

    let insertTail x =
        let rec aux acc =
```

```
            function
            | [] -> (x :: acc) |> List.rev
            | y :: ys when y < x -> aux (y :: acc) ys
            | y :: ys            -> (List.rev (y :: x

        aux []

    let insertionSortTail lst =
        let rec aux acc =
            function
            | [] -> acc
            | x :: xs -> aux (insertTail x acc) xs

        aux [] lst

(* Question 1.3 *)

    (*
    Q: Why are the higher-order functions from the Lis
    not a good fit to implement insert?

    A: Because higher-order functions from the list li
       terminates as soon as it has found the insertio
       these higher-order functions to write insert, i

       A few examples that use higher-order functions
       but are in no way required for the exam.
    *)

    let insertFilter x lst = List.filter ((>=) x) lst
    let insertFold x lst =
        List.foldBack
            (fun y ->
                function
                | (true, acc) when x >= y -> (false, y
                | (cont, acc)             -> (cont, y
```

```
            lst
            (true, []) |>
            snd

    let insertFold2 x lst =
        List.foldBack
            (fun y (cont, acc) ->
                if cont && x >= y then
                    (false, y :: x :: acc)
                else
                    (cont, y :: acc))
            lst
            (true, []) |>
            snd



    let insertionSort2 lst = List.foldBack insertTail

(* Question 1.4 *)

    let insertBy f x =
        let rec aux acc =
            function
            | [] -> (x :: acc) |> List.rev
            | y :: ys when f y < f x -> aux (y :: acc)
            | y :: ys            -> (List.rev (y :: x

        aux []

    let insertionSortBy f lst = List.foldBack (insertB
```

Question 2

Vores svar

```
(* 2: Code Comprehension *)
    let rec foo x =
```

```
        function
        | y :: ys when x = y -> ys
        | y :: ys            -> y :: (foo x ys)

    let rec bar x =
        function
        | []        -> []
        | xs :: xss -> (x :: xs) :: bar x xss

    let rec baz =
        function
        | [] -> []
        | [x] -> [[x]]
        | xs  ->
            let rec aux =
                function
                | []      -> []
                | y :: ys -> ((foo y >> baz >> bar y)
            aux xs

    let super y xs = (foo y >> baz >> bar y) xs

(* Question 2.1 *)

    (*

    Q: What are the types of functions foo,  bar, and

    A:
        foo: 'a -> 'a list -> 'a list
        bar: 'a -> 'a list list -> 'a list list
        baz: 'a list -> 'a list list



    Q: What do functions foo, bar, and baz do?
       Focus on what they do rather than how they do i
```

```
    A:
        foo: takes one parameter, and removes the firs
             the list
             e.g. foo 1 [1;2;3] = [2;3]

        bar: Takes one parameter, and puts it in front
             e.g. bar 1 [[2];[3];[4]] = [[1;2];[1;3];

        baz: Takes a list as input, and "scrambles" it
             different combinations that exists
             e.g. baz [1;2;3] = [[1;2;3];[2;3;1];[3;2


    Q: What would be appropriate names for functions
       foo, bar, and baz?

    A:
        foo = removeSingle
        bar = prependOnLists
        baz = getCombinations

    *)


(* Question 2.2 *)


    (*
    The function foo generates a warning during compil
    Warning: Incomplete pattern matches on this expres


    Q: Why does this happen, and where?

    A: It happens in the "function" match case, due to
```

```
        matching on an empty list.

    Q: For these particular three functions will this
       pattern match ever cause problems for any possi
       If yes, why; if no, why not.

    A: No, due to baz matching [] -> [], so no empty l
       This can also be seen as it matches xs (somethi
       sends that xs to foo.

    *)

    let rec foo2 x =
        function
        | [] -> []
        | y :: ys when x = y -> ys
        | y :: ys            -> y :: (foo2 x ys)


(* Question 2.3 *)

    (*
    In the function baz there is a sub expression foo

    Q: What is the type of this expression

    A: 'a -> 'a list -> 'a list list

    Q: What does it do? Focus on what it does rather t

    A: It takes an input parameter, and finds all comb
    this parameter. The list MUST contain the paramete
    of the parameter in the list, and places the param

    *)
```

```
(* Question 2.4 *)

    let bar2 x xs = List.map (fun y -> x::y) xs

(* Question 2.5 *)

    let rec baz2 =
        function
        | [] -> []
        | [x] -> [[x]]
        | xs  -> List.fold (fun acc elem -> (foo elem

(* Question 2.6 *)

    (*

    Q: The function foo is not tail recursive. Why?

    A:
        foo 3 [1;2;3]
        1 :: (foo 3 [2;3]
        1 :: (2 :: (foo 3 [3]))
        1 :: (2 :: ([]))        -> recursion ends, ret
        1 :: [2]
        [1;2]

    *)

    let fooTail y lst =
        let rec aux xs c =
            match xs with
            | [] -> c []
            | x' :: xs' when x' = y -> c xs'
            | x' :: xs' -> aux xs' (fun a -> c (x' ::
        aux lst id
```

▼ Jesper svar

```
module Question2

(* 2: Code Comprehension *)
    let rec foo x =
        function
        | y :: ys when x = y -> ys
        | y :: ys            -> y :: (foo x ys)

    let rec bar x =
        function
        | []        -> []
        | xs :: xss -> (x :: xs) :: bar x xss

    let rec baz =
        function
        | [] -> []
        | [x] -> [[x]]
        | xs  ->
            let rec aux =
                function
                | []      -> []
                | y :: ys -> ((foo y >> baz >> bar y)
            aux xs

    (*

    Q: What are the types of functions foo,  bar, and

    A:  foo : 'a -> 'a list -> 'a list when 'a : equal
        bar : 'a -> 'a list list -> 'a list list
        baz : 'a list -> 'a list list when 'a : equali
```

```
    Q: What do functions foo, bar, and baz do?

    A: foo takes an element `x` and a list `lst` and r
       bar taks an element `x` and a list of lists `ls
       baz takes a list `lst` and returns a list conta


    Q: What would be appropriate names for functions

    A: foo: removeFirstOccurrence
       bar: addToAll
       baz: permutations


    *)

(* Question 2.2 *)

    (*
    The function foo generates a warning during compil
    Warning: Incomplete pattern matches on this expres


    Q: Why does this happen, and where?

    A: It happens because foo does not have a case tha
       In particular, this case will be reached whenev
       that is not in that list.

    Q: For these particular three functions will this
       pattern match ever cause problems for any possi
       If yes, why; if no, why not.

    A: No. The empty list case for foo will only ever
       from a list that does not exist in the list. Th
       and removes elements from that list using foo -
```

```
        that does not exist in that list.


    *)


    let rec foo2 x =
        function
        | []                -> []
        | y :: ys when x = y -> ys
        | y :: ys            -> y :: (foo x ys)


(* Question 2.3 *)

    (*
    In the function baz there is a sub expression foo

    Q: What is the type of this expression

    A: 'a list -> 'a list list when 'a : equality


    Q: What does it do? Focus on what it does rather t

    A: The function takes a list `lst` and returns al

    *)

(* Question 2.4 *)

    let bar2 x lst = List.map (fun xs -> x :: xs) lst

(* Question 2.5 *)

    let rec baz2 =
```

```
        function
        | [] -> []
        | [x] -> [[x]]
        | xs  -> List.foldBack (fun y acc -> (foo y >>

(* Question 2.6 *)

    (*

    Q: The function foo is not tail recursive. Why?

    A: The function foo is not called in a tail-recurs
       be generated that cannot be reduced until the b

       foo 3 [1;2;3;3;4;5] ->
       1 :: (foo 3 [2;3;4;5]) ->
       1 :: 2 :: (foo 3 [3;3;4;5]) ->
       1 :: 2 :: [3; 4; 5] ->
       [1; 2; 3; 4; 5]

    *)

    let fooTail x lst =
        let rec aux c =
            function
            | [] -> c []
            | y :: ys when x = y -> c ys
            | y :: ys -> aux (fun result -> c (y :: re

        aux id lst
```

Question 3

vores svar

```
(* Question 3.1 *)
```

```
    type shape = Rock | Paper | Scissor
    type result = P1Wins | P2Wins | Draw

    let mkShape s =
        match s with
        | "rock" -> Rock
        | "paper" -> Paper
        | "scissors" -> Scissor

    let resultToString r =
        match r with
        | P1Wins -> "playerOneWin"
        | P2Wins-> "playerTwoWin"
        | Draw-> "draw"

    let shapeToString s =
        match s with
        | Rock -> "rock"
        | Paper -> "paper"
        | Scissor -> "scissors"

    let rps s1 s2 =
        match s1, s2 with
        | Rock, Paper | Scissor, Rock | Paper, Scisso
        | Paper, Rock | Rock, Scissor | Scissor, Pape
        | _, _ -> Draw

(* Question 3.2 *)

    type strategy = (shape * shape) list -> shape

    let parrot s moves =
        match moves with
        | (_, s2) :: _ -> s2
        | _ -> s
```

```
    let beatingStrat moves =
        let opponentMoves = List.map snd moves
        let numRocks = opponentMoves |> List.filter (
        let numScissors = opponentMoves |> List.filter
        let numPapers = opponentMoves |> List.filter (

        if numScissors >= numPapers && numScissors >=
            Rock
        elif numRocks >= numPapers && numRocks >= numS
            Paper
        else
            Scissor

    let roundRobin lst =
        let mutable temp = lst
        let rec aux () =
            match temp with
            | [] ->
                temp <- lst
                aux ()
            | x :: xs ->
                temp <- xs
                x
        fun _ -> aux ()

(* Question 3.3 *)

    (*

    Q: It may be tempting to generate a function that
       point tuple after n rounds and then use Seq.ini
       generate the sequence. This is not a good solut

    A:
        Seq.initInfinite only works well when used wit
        previous entries in the sequence to calculate
```

```
        case, then for the nth game, all games prior t
        then recomputed when moving on to game (n+1).

    *)

    let bestOutOf strat1 strat2 =
        let unfolder = (fun (p1moves, p2moves, p1, p2)
            let s1 = strat1 p1moves
            let s2 = strat2 p2moves
            let (p1', p2') =
                match rps s1 s2 with
                | P1Wins -> (p1 + 1, p2)
                | P2Wins -> (p1, p2 + 1)
                | Draw-> (p1, p2)
            Some ((p1', p2'), ((s1, s2) :: p1moves, (s
        Seq.unfold unfolder ([], [], 0, 0)
        |> Seq.append (Seq.singleton (0, 0))

(* Question 3.4 *)

    let playTournament rounds players =

        let rec initRound acc =
            function
            | [] -> (acc, [])
            | [x] -> (acc, [x])
            | x :: y :: xs -> initRound ((x, y) :: acc

        let rec aux =
            function
            | [] -> None
            | [(_, id)] -> Some id
            | players ->
                let (pairs, rest) = initRound [] playe

                pairs |>
```

```
                  List.map
                      (fun ((p1, id1), (p2, id2)) ->
                          async {
                              let (p1win, p2win) = best(
                              return
                                  if p1win = p2win
                                      then None
                                  elif p1win > p2win
                                      then Some (p1, id1
                                  else
                                      Some (p2, id2)
                          }) |>
                      Async.Parallel |>
                      Async.RunSynchronously |>
                      Array.toList |>
                      List.filter Option.isSome |>
                      List.map Option.get |>
                      (fun lst -> aux (lst @ rest))
          aux (List.mapi (fun i x -> (x, i)) players)
```

▼ Jesper svar

```
module Question3

(* 3: Rock Paper Scissors *)

(* Question 3.1 *)

    type shape =  Rock | Paper | Scissors
    type result = P1Win | P2Win | Draw



    let mkShape =
       function
       | "rock"     -> Rock
       | "paper"    -> Paper
```

```
        | "scissors" -> Scissors
        | s          -> failwith (sprintf "invalid shap


    let shapeToString =
       function
        | Rock     -> "rock"
        | Paper    -> "paper"
        | Scissors -> "scissors"

    let resultToString =
        function
        | P1Win -> "playerOneWin"
        | P2Win -> "playerTwoWin"
        | Draw  -> "draw"


    let rps s1 s2 =
        match s1, s2 with
        | Rock, Paper     -> P2Win
        | Paper, Scissors -> P2Win
        | Scissors, Rock  -> P2Win
        | _, _            -> if s1 = s2 then Draw else

(* Question 3.2 *)

    type strategy = (shape * shape) list -> shape

    let parrot s =
        function
        | [] -> s
        | (_, m) :: _ -> m

    let beatingStrat moves =
        let opponentMoves = List.map snd moves
        let numRocks    = opponentMoves |> List.filte
```

```
    let numPapers   = opponentMoves |> List.filter
    let numScissors = opponentMoves |> List.filter

    if numScissors >= numPapers && numScissors >=
        Rock
    else if numRocks >= numPapers && numRocks >= n
        Paper
    else
        Scissors

(* Alternative version *)

let beats =
    function
    | Rock -> Paper
    | Paper -> Scissors
    | Scissors -> Rock

let beatingStrat2 : strategy =
    List.map snd >>
    List.countBy id >>
    List.sortByDescending snd >>
    (function
     | [] -> [(Paper, 0)]
     | (_, max)::_ as lst -> List.filter (snd >> (
    List.map (fst >> beats) >>
    List.sort >>
    List.head

let roundRobin lst =
    let mutable temp = lst
    let rec aux () =
        match temp with
        | [] -> temp <- lst; aux ()
        | x :: xs -> temp <- xs; x
```

```
        fun _ -> aux ()

(* Question 3.3 *)

    (*

    Q: It may be tempting to generate a function that
       point tuple after n rounds and then use Seq.ini
       generate the sequence. This is not a good solut

    A: Seq.initInfinite only works well when used with
       to calculate its elements. If you use it in thi
       must be computed, and then recomputed when movi

    *)

    let bestOutOf strat1 strat2 =
        Seq.unfold
            (fun (p1moves, p2moves, p1, p2) ->
                let s1 = strat1 p1moves
                let s2 = strat2 p2moves
                let (p1', p2') =
                    match rps s1 s2 with
                    | P1Win -> (p1 + 1, p2)
                    | P2Win -> (p1, p2 + 1)
                    | Draw  -> (p1, p2)
                Some ((p1', p2'), ((s1, s2)::p1moves,

            ([], [], 0, 0) |>

        Seq.append (Seq.singleton (0, 0))


(* Question 3.4 *)
```

```
let playTournament rounds players =

    let rec initRound acc =
        function
        | [] -> (acc, [])
        | [x] -> (acc, [x])
        | x::y::xs -> initRound ((x, y)::acc) xs

    let rec aux =
        function
        | [] -> None
        | [(_, id)] -> Some id
        | players ->

            let (pairs, rest) = initRound [] playe

            pairs |>
            List.map
                (fun ((p1, id1), (p2, id2)) ->
                    async {
                        let (p1win, p2win) = best(
                        return if p1win = p2win th

                    }) |>
            Async.Parallel |>
            Async.RunSynchronously |>
            Array.toList |>
            List.filter Option.isSome |>
            List.map Option.get |>
            (fun lst -> aux (lst @ rest))

    aux (List.mapi (fun i x -> (x, i)) players)
```

▼ Question 4

▼ vores svar

```
```

- **jesper svar**

Thor's svar (ikke færdig)

```
                module Exam2020
(* If you are importing this into F# interactive then co
   the line above and remove the comment for the line be

   Do note that the project will not compile if you do t
   it does allow you to work in interactive mode and you
   to make the project compile work again.

   Do not remove the line (even though that does work) b
   introduce indentation errors in your code that may be
   to switch back to project mode.

   Alternative, keep the line as is, but load ExamIntera
   *)
(* module Exam2020 = *)

(* 1: Insertion sort *)

(* Question 1.1 *)

    let rec insert (x: 'a) (lst: 'a list) =
        match lst with
        | [] -> [x]
        | head :: tail -> if x > head then
                                head :: insert x tail
                            else
                                x :: insert head tail

    let rec insertAlt (x: 'a) (lst: 'a list) =
        match lst with
        | [] -> [x]
```

```
        | head :: tail -> min head x :: (insert (max hea

    insert 5 [1; 3; 8; 9]

    let rec insertionSort (lst: 'a list) =
        match lst with
        | [] -> []
        | head :: tail -> insert head (insertionSort tai

    insertionSort [5; 3; 1; 8; 9]

    let insertionSortHighOrder (lst: 'a list) =
        List.foldBack(fun value acc -> insert value acc)

    let insertionSortHighOrderAlt (lst: 'a list) = List.

    insertionSort [5; 3; 1; 8; 9]

(* Question 1.2 *)

    let insertTail (x: 'a) (lst: 'a list) =
        let rec aux (x: 'a) (lst: 'a list) acc =
            match lst with
            | [] -> acc @ [x]
            | head :: tail -> aux (max head x) tail (acc
        aux x lst []

    insertTail 5 [1; 3; 8; 9]

    let insertionSortTail (lst: 'a list) =
        let rec aux (lst: 'a list) acc =
            match lst with
            | [] -> acc
            | head :: tail -> aux tail (insertTail head
        aux lst []
```

```
    insertionSortTail [5; 3; 1; 8; 9]

(* Question 1.3 *)

    (*
    Q: Why are the higher-order functions from the List
    not a good fit to implement insert?

    A: <Your answer goes here>
    Higher-order function does not allow you stop early
    When you normally recurse through a list, you can "s
    With higher-order functions you always have to recur
    *)

    let insertionSort2 (lst: 'a list) =
        List.foldBack(fun value acc -> insertTail value

    let insertionSort2B (lst: 'a list) =
        List.fold(fun acc value -> insertTail value acc)


(* Question 1.4 *)

    let insertBy (f: 'a -> 'b) (x: 'a) (lst: 'a list) =
        let rec aux lst acc =
            match lst with
            | [] -> x :: acc
            | head :: tail when f x < f head -> (List.re
            | head :: tail -> acc //TODO FIX
        aux lst []

    let insertionSortBy (f: 'a -> 'b) (lst: 'a list) =
        List.fold(fun acc value -> insertBy f value lst)

(* 2: Code Comprehension *)
    let rec foo x = //Removes first instance of x from t
```

```
        function
        | y :: ys when x = y -> ys
        | y :: ys            -> y :: (foo x ys)

    let test2 = foo 1 [2;1;2;1;3]

    let rec bar x = //adds x to the front of all lists i
        function
        | []        -> []
        | xs :: xss -> (x :: xs) :: bar x xss

    bar 1 [[1;2];[3;2];[3;4;2]]

    let rec baz = //Creates a list of lists containing l
        function
        | [] -> []
        | [x] -> [[x]]
        | xs  ->
            let rec aux =
                function
                | []      -> []
                | y :: ys -> ((foo y >> baz >> bar y) xs
            aux xs

    let test3 = baz [1;2;3]

(* Question 2.1 *)

    (*

    Q: What are the types of functions foo,  bar, and ba

    A: <Your answer goes here>


    Q: What do functions foo, bar, and baz do?
```

```
    Focus on what they do rather than how they do it.

A: <Your answer goes here>


Q: What would be appropriate names for functions
    foo, bar, and baz?

A: <Your answer goes here>
    remove
    addToAll
    permutationsOf

*)


(* Question 2.2 *)


    (*
    The function foo generates a warning during compilat
    Warning: Incomplete pattern matches on this expressi


    Q: Why does this happen, and where?

    A: <Your answer goes here>
    We need to complete the pattern with _ -> []


    Q: For these particular three functions will this in
       pattern match ever cause problems for any possibl
       If yes, why; if no, why not.

    A: <Your answer goes here>
    It would cause an issue if you for example, try foo
```

```
    but the baz method can be called like baz [1;2;3;4]
    and it will only call the foo function with numbers
    so the foo function will not cause any problems in b

    *)

    let rec foo2 x = //Removes first x from the list, wh
        function
        | y :: ys when x = y -> ys
        | y :: ys            -> y :: (foo2 x ys)
        | _ -> []

    let testFoo2 = foo2 1 [2;1;2;3]

(* Question 2.3 *)

    (*
    In the function baz there is a sub expression foo y

    Q: What is the type of this expression

    A: <Your answer goes here>
    ">>" Composes two functions (forward composition ope


    Q: What does it do? Focus on what it does rather tha

    A: <Your answer goes here>
    Using function composition it takes the first functi
    and sends that result to the next function "baz" whi
    finally send the result to the last function "bar y"

    "foo y" creates a function that will remove y from t
    this is sent to baz so it creates the unique list co
    Then after that, y is re-added using "bar y" to thos
    This will finally create all the possible combinatio
```

```
    *)

(* Question 2.4 *)

    let bar2 (x: 'a) (bigList: 'a list list) =
        List.map(fun innerList -> //List.map is a higher
                x :: innerList
            ) bigList

    let rec barOld x = //adds x to the front of all list
        function
        | []        -> []
        | xs :: xss -> (x :: xs) :: bar x xss


(* Question 2.5 *)

    let rec baz2 (x: 'a list) =
        match x with
        | [] -> []
        | [x] -> [[x]]
        | xs  ->
            List.foldBack(fun valu acc ->
                ((foo valu >> baz >> bar valu) xs) @ acc
            ) xs List.Empty
//            List.fold(fun acc valu ->
//                ((foo valu >> baz >> bar valu) xs) @ a
//            ) List.Empty xs

    let rec bazOld = //Creates a list of lists containin
        function
        | [] -> []
        | [x] -> [[x]]
        | xs  ->
            let rec aux =
```

```
                function
                | []      -> []
                | y :: ys -> ((foo y >> baz >> bar y) xs
            aux xs

(* Question 2.6 *)

    (*

    Q: The function foo is not tail recursive. Why?

    A: <Your answer goes here>

    *)

    let fooTail (x: 'a) (lst: 'a list) =

        let rec aux (x: 'a) (lst: 'a list) cont =
            match lst with
            | [] -> cont []
            | y :: ys when x = y -> cont ys
            | y :: ys            -> aux (fun res -> cont
        aux x lst id

    let fooTail x lst =
        let rec aux c =
            function
            | [] -> c []
            | y :: ys when x = y -> c ys
            | y :: ys -> aux (fun result -> c (y :: resu
        aux id

    let rec fooOld x = //Removes x from the list, while
        function
        | y :: ys when x = y -> ys
        | y :: ys            -> y :: (foo x ys)
```

```
(* 3: Rock Paper Scissors *)

(* Question 3.1 *)

    type shape =
        | Rock
        | Paper
        | Scissors

    type result =
        | Draw
        | PlayerOneWins
        | PlayerTwoWins

    let rps (s1: shape) (s2: shape) =
        match s1, s2 with
        | shape.Paper, shape.Rock -> result.PlayerOneWin
        | shape.Rock, shape.Scissors -> result.PlayerOne
        | shape.Scissors, shape.Paper -> result.PlayerOn
        | sh1, sh2 when sh1 = sh2  -> result.Draw
        | _ -> result.PlayerTwoWins


(* Question 3.2 *)

    type strategy = (shape * shape) list -> shape

    let parrot (s1: shape) (moves: (shape * shape) list)
        match moves with
        | [] -> s1
        | head :: tail -> snd head

    let beatingStrat (moves: (shape * shape) list) =
        let oppMoves = List.map snd moves
        let numRocks = oppMoves |> List.filter (fun s ->
```

```
        let numPapers = oppMoves |> List.filter (fun s -
        let numScissors = oppMoves |> List.filter (fun s

        if (numScissors >= numPapers && numScissors >= n
            shape.Rock
        else if numRocks >= numPapers && numRocks >= num
            shape.Paper
        else
            shape.Scissors

    beatingStrat [(shape.Scissors, shape.Paper)]

    beatingStrat [(shape.Scissors, shape.Paper); (shape.

    let roundRobin _ = failwith "not implemented"

(* Question 3.3 *)

    (*

    Q: It may be tempting to generate a function that ca
       point tuple after n rounds and then use Seq.initI
       generate the sequence. This is not a good solutio

    A: <Your answer goes here>

    *)

    let bestOutOf _ = failwith "not implemented"

(* Question 3.4 *)

    let playTournament _ = failwith "not implemented"

(* 4: Revers Polish Notation *)
```

```
(* Question 4.1 *)

    type stack = int list
    let emptyStack: stack = []

(* Question 4.2 *)

    type SM<'a> = S of (stack -> ('a * stack) option)

    let ret x = S (fun s -> Some (x, s))
    let fail  = S (fun _ -> None)
    let bind f (S a) : SM<'b> =
        S (fun s ->
            match a s with
            | Some (x, s') ->
                let (S g) = f x
                g s'
            | None -> None)

    let (>>=) x f = bind f x
    let (>>>=) x y = x >>= (fun _ -> y)

    let evalSM (S f) = f emptyStack

    let push (x: 'a) =
        S(fun stak -> Some(((), x :: stak)))

    let pop =
        S(fun stak ->
            if stak.IsEmpty then
                None
            else
                Some(stak.Head, stak.Tail))

    push 5 >>>= push 6 >>>= pop |> evalSM
    pop |> evalSM
```

```
(* Question 4.3 *)

    let write str : SM<unit> = S (fun s -> printf "%s" s

    let read =
        let rec aux acc =
            match System.Console.Read() |> char with
            | '\n' when acc = [] -> None
            | c    when System.Char.IsWhiteSpace c ->
                acc |> List.fold (fun strAcc ch -> (stri
            | c -> aux (c :: acc)

        S (fun s -> Some (aux [], s))

    (*

    Q: Consider the definition of write There is a reaso
       is S (fun s -> printf "%s" str; Some ((), s)) and
       ret (printf "%s" str). For a similar reason, in r
       S (fun s -> Some (aux [], s)) and not ret (aux []
       What is the problem with using ret in both of the

    A: <Your answer goes here>

    *)

(* Question 4.4 *)

    (* You may solve this exercise either using monadic
        using computational expressions. *)

    type StateBuilder() =
```

```
        member this.Bind(f, x)     = bind x f
        member this.Return(x)      = ret x
        member this.ReturnFrom(x) = x
        member this.Combine(a, b) = a >>= (fun _ -> b)

    let state = new StateBuilder()

    let calculateRPN _ = failwith "not implemented"
```

Exam 2019-08-13

Question 1

Vores svar

```
module Question1

    type Sum<'A, 'B> =
    | Left of 'A
    | Right of 'B

    (* Question 1.1 *)

    let sum1: Sum<int list, bool option> = Left [0;1;2
    let sum2: Sum<int list, bool option> = Right (Some

    let sumMap f g sum =
        match sum with
        | Left x -> f x
        | Right y -> g y

    (* Question 1.2 *)

    type SumColl<'A, 'B> =
    | Nil
    | CLeft of 'A * SumColl<'A, 'B>
    | CRight of 'B * SumColl<'A, 'B>
```

```
let sumColl: SumColl<bool list, int> = CLeft ([tru

let rec ofList (sum: Sum<'A, 'B> list): SumColl<'a
    match sum with
    | head :: tail ->
        match head with
        | Left x -> CLeft (x, (ofList tail))
        | Right y -> CRight (y, (ofList tail))
    | [] -> Nil

(* Question 1.3 *)

let reverse (sum: SumColl<'a, 'b>): SumColl<'a, 'b
    let rec aux (s: SumColl<'a, 'b>) acc =
        match s with
        | Nil -> acc
        | CLeft (a, b) -> aux b (CLeft (a, acc))
        | CRight (a, b) ->  aux b (CRight (a, acc)
    aux sum Nil

(* Question 1.4 *)

let ofList2 (sum: Sum<'A, 'B> list) =
    List.foldBack (fun (elem: Sum<'A, 'B>) (acc: S
        match elem with
        | Left x -> CLeft (x, acc)
        | Right y -> CRight (y, acc)
        ) sum Nil

(* Question 1.5 *)

let rec foldBackSumColl f g sum acc =
    match sum with
    | CLeft (a, b) -> f a (foldBackSumColl f g b a
```

```
        | CRight (a, b) -> g a (foldBackSumColl f g b
        | Nil -> acc
```

▼ Jespers svar

```
module Question1Solution
    type Sum<'A, 'B> =
    | Left of 'A
    | Right of 'B

    (* Question 1.1 *)

    let sum1 = Left [5]
    let sum2 = Right (Some true)

    let sumMap f g =
        function
        | Left a -> f a
        | Right b -> g b

    (* Question 1.2 *)

    type SumColl<'A, 'B> =
    | Nil
    | CLeft of 'A * SumColl<'A, 'B>
    | CRight of 'B * SumColl<'A, 'B>

    let sumColl = CLeft ([true], CRight (42, Nil))

    let rec ofList =
        function
        | [] -> Nil
        | Left a :: ss -> CLeft (a, ofList ss)
        | Right b :: ss -> CRight (b, ofList ss)

    (* Question 1.3 *)
```

```
    let reverse lst =
        let rec aux acc =
            function
            | Nil -> acc
            | CLeft (a, ss) -> aux (CLeft (a, acc)) ss
            | CRight (b, ss) -> aux (CRight (b, acc))

        aux Nil lst

    (* Question 1.4 *)

    let lcons ss a = CLeft(a, ss)
    let rcons ss b = CRight(b, ss)

    let ofList2 lst =
        List.foldBack (fun x acc -> sumMap (lcons acc)

    (* Question 1.5 *)

    let rec foldBackSumColl f g =
        function
        | Nil -> id
        | CLeft (a, ss)  -> f a << foldBackSumColl f g
        | CRight (b, ss) -> g b << foldBackSumColl f g
```

Question 2

Thor's svar

```
module CodeComprehension

    let f s =
        let l = String.length s
        let rec aux =
            function
            | i when i = l -> []
```

```
        | i -> s.[i] :: aux (i + 1)

    aux 0

let g s =
    s |> f |>
    List.filter System.Char.IsLetter |>
    List.map System.Char.ToLower |>
    fun lst -> lst = List.rev lst

(* Question 2.1 *)

(*
What are the types of functions f and g?
f: string -> char list
g: string -> bool

What do functions f and g do? Focus on what they d
f: Converts a string to a char list
g: Returns true if string does not contain letters
   The list filters away all chars that are not le
   Then it makes them lowercase
   Then it checks if the list is equal to the reve

What would be appropriate names for functions f an
f: stringToCharList
g: doesNotContainLettersOrPalindrome

*)

(* Question 2.2 *)

let f2 (str: string) = [ for chr in str -> chr ]
let f2Alt (str: string) = [ for i in 0 .. (String

(* Question 2.3 *)
```

```
let g2 =
    f >>
    List.filter System.Char.IsLetter >>
    List.map System.Char.ToLower >>
    fun lst -> lst = List.rev lst

(* Question 2.4 *)

(*

f "Hell"
Gets char at index i of string given
(aux 0)
'H' :: (aux 1)
'H' :: 'E' :: (aux 2)
'H' :: 'E' :: 'L' :: (aux 3)
'H' :: 'E' :: 'L' :: 'L' :: (aux 4)
'H' :: 'E' :: 'L' :: 'L' :: []
'H' :: 'E' :: 'L' :: ['L']
'H' :: 'E' :: ['L'; 'L']
'H' :: ['E'; 'L'; 'L']
['H'; 'E'; 'L'; 'L']
Finally a char list

<Your answers go here>
*)

let fTail s =
    let l = String.length s

    let rec aux (i: int) cont =
        match i with
        | i when i = l -> cont []
        | i -> aux (i + 1) (fun lst -> cont(s.[i]
```

```
        aux 0 id

    fTail "Hello"

    (* Question 2.5 *)

    let rec gOpt (highStr: string) =
        let str = highStr.ToLower()

        let rec aux (start: int) (slut: int) =
            match start, slut with
            | (x, y) when x = y || (abs x - y) = 1 ->
            | x, y when System.Char.IsLetter str.[x] &
            | x, _ when System.Char.IsLetter str.[x] =
            | _, y when System.Char.IsLetter str.[y] =
            | _ -> aux (start + 1) (slut - 1)

        aux 0 (str.Length - 1)

        gOpt "Dromedaren Alpotto planerade mord!!!"
```

▼ Jesper's svar

```
module CodeComprehensionSolution

    let f s =
        let l = String.length s
        let rec aux =
            function
            | i when i = l -> []
            | i -> s.[i] :: aux (i + 1)

        aux 0

    let g s =
        s |> f |>
```

```
    List.filter System.Char.IsLetter |>
    List.map System.Char.ToLower |>
    fun lst -> lst = List.rev lst

(* Question 2.1 *)

(*
f has type string -> char list
g has type string -> bool

f takes a string s and returns a list with all cha
g takes a string s and returns true if s is a pali
characters that are not letters and is case insens

An appropriate name for f would be explode or stri
An appropriate name for g would be isPalindrome
*)

(* Question 2.2 *)


let f2 s = [for c in s do yield c]

(* Question 2.3 *)

let g2 =
    f >>
    List.filter System.Char.IsLetter >>
    List.map System.Char.ToLower >>
    fun lst -> lst = List.rev lst

(* Question 2.4 *)

(*

The inner function aux of f is not tail recursive
```

```
f "ITU" -->
let l = 3
let rec aux =
    function
    | i when i = 3 -> []
    | i -> "ITU".[i] :: aux (i + 1)

aux 0 -->

"ITU".[0] :: aux (0 + 1) -->
'I' :: aux 1 -->

'I' :: "ITU".[1] :: aux (1 + 1) -->
'I' :: 'T' :: aux 2 -->

'I' :: 'T' :: "ITU".[2] :: aux (2 + 1) -->
'I' :: 'T' :: 'U' :: aux 3 -->

'I' :: 'T' :: 'U' :: [] -->
['I'; 'T'; 'U']

The computation produces a long list of cons-opera
has been reached.

*)

let fTail s =
    let l = String.length s
    let rec aux c =
        function
        | i when i = l -> c []
        | i -> aux (fun result -> c (s.[i]::result

    aux id 0
(* Question 2.5 *)
```

```
    let gOpt (s : string) =
        let rec aux i =
            function
            | j when i >= j -> true
            | j when not (System.Char.IsLetter s.[i])
                 aux (i + 1) j
            | j when not (System.Char.IsLetter s.[j])
                aux i (j - 1)
            | j when System.Char.ToLower s.[i] = Syste
                aux (i + 1) (j - 1)
            | _ -> false

        aux 0 (String.length s - 1)
```

▼ Question 3

▼ Vores svar

```
module Question3

    (* Question 3.1 *)

    let foo f =
        let mutable m = Map.empty
        let aux x =
            match Map.tryFind x m with
            | Some y -> y
            | None ->
                m <- Map.add x (f x) m; f x
        aux

    let rec bar x : float=
        match x with
        | 0 | 1 -> 1.0
```

```
    | y -> baz (y - 1) + baz (y - 2)
and baz = foo bar

let calculateGoldenRatio (n: int) =
    (baz (n+1)) / (baz n )

(* Question 3.2 *)

let grSeq = Seq.unfold(fun i -> Some (calculateGol

(* Question 3.3 *)

let goldenRectangleSeq (x: float) = Seq.unfold (fu

let goldenTriangleSeq (b: float) =
    Seq.unfold (fun i ->
        let gRatio = Seq.item i grSeq
        let height = b * sqrt(gRatio * gRatio - (1
        let result = (b * height) / 2.0

        Some (result, i+1)
    ) 0

(* Question 3.4 *)

let goldenRectangleTriangle (b: float) =
    Seq.unfold (fun i ->
            let gRatio = Seq.item i grSeq
            let height = b * sqrt(gRatio * gRatio
            let resultTriangle = (b * height) / 2
            let resultSquare = (b * (b * Seq.item

            Some ((resultSquare, resultTriangle),
        ) 0
```

▼ Jespers svar

```
module Question3Solution

    (* Question 3.1 *)

    let calculateGoldenRatio x =
        let rec aux a b =
            function
            | 0 -> float b / float a
            | n -> aux b (a + b) (n - 1)

        aux 1 1 x

    (* Question 3.2 *)

    let grSeq = Seq.unfold (fun (a, b) -> Some (b / a,

    (* Question 3.3 *)

    let goldenRectangleSeq x =
        Seq.map (fun i -> x * x * i) grSeq

    let goldenTriangleSeq b =
        let height i = b * System.Math.Sqrt (i * i - 
        Seq.map (fun x -> b * height x / 2.0) grSeq

    (* Question 3.4 *)

    let goldenRectangleTriangle b =
        let rec aux s1 s2 =
            seq {
                yield (Seq.head s1, Seq.head s2)
                yield! aux (Seq.tail s1) (Seq.tail s2
            }
```

```
        aux (goldenRectangleSeq b) (goldenTriangleSeq
```

Question 4

Thor's svar

```
module Question4

    (* Question 4.1 *)
    type color =
       | Red
       | Green
       | Blue
       | Purple
       | Orange
       | Yellow

    type shape =
       | Square
       | Circle
       | Star
       | Diamond
       | Club
       | Cross

    type tile = color * shape

    let getColor (color: string) =
        match color with
        | "red" -> Red
        | "green" -> Green
        | "blue" -> Blue
        | "purple" -> Purple
        | "orange" -> Orange
        | _ -> Yellow
```

```
let getShape (shape: string) =
    match shape with
    | "square" -> Square
    | "circle" -> Circle
    | "star" -> Star
    | "diamond" -> Diamond
    | "club" -> Club
    | _ -> Cross

let mkTile (color: string) (shape: string): tile =

mkTile "blue" "diamond"
(* Question 4.2 *)
let tileToString (t: tile) =
    let color = fst t
    let shape = snd t
    color.ToString().ToLower() + " " + shape.ToStr

tileToString (mkTile "blue" "diamond")

let sameColors (ts: tile list) =
    if List.isEmpty ts then false
    else

    let firstColor = fst ts.Head

    let rec aux (ts: tile list) =
        match ts with
        | [] -> true
        | head :: tail -> if fst head = firstColor
    aux ts


let rec validTilesAlt (ts: tile list) (t: tile) =

    let sameColor = sameColors ts
```

```
    let colorT, shapeT = t

    List.fold(fun acc (colorH, shapeH) ->

                    if sameColor then
                        if colorT <> colorH || sha
                            false else acc
                    else
                        if shapeT <> shapeH || col
                            false else acc

                ) true ts


let rec validTiles (ts: tile list) (t: tile) =

    let sameColor = sameColors ts
    let colorT, shapeT = t

    let rec aux (ts: tile list) =
        match ts with
        | [] -> true
        | head :: tail ->
            let colorH, shapeH = head

            if sameColor then
                if colorT <> colorH || shapeT = sh
                    false
                else aux tail
            else
                if shapeT <> shapeH || colorT = co
                    false
                else aux tail
    aux ts

let bc = mkTile "blue" "club"
```

```
let bd = mkTile "blue" "diamond"
let rd = mkTile "red" "diamond"
let pd = mkTile "purple" "diamond"
let ps = mkTile "purple" "star"
let pc = mkTile "purple" "club"

validTiles [bd; rd] pd
validTiles [bd; rd] bd
validTiles [ps; pd] pc
validTiles [ps; pd] ps

validTiles [ps; pd] bc

sameColors [ps; pd; ps]
sameColors [ps; pd; bc]



(* Question 4.3 optional *)

type coord = Coord of int * int
type board = Board of Map<coord, tile>
type direction = Left | Right | Up | Down

let moveCoord (Coord(x, y)) (d: direction): coord
    match d with
    | Left -> Coord(x - 1, y)
    | Right -> Coord(x + 1, y)
    | Up -> Coord(x, y - 1)
    | Down -> Coord(x, y + 1)

moveCoord (Coord (0, 0)) Left
moveCoord (Coord (0, 0)) Right
moveCoord (Coord (0, 0)) Up
moveCoord (Coord (0, 0)) Down
```

```
let collectTiles (Board b) c d =
    let rec aux (c: coord) (lst: tile list) =
        match Map.tryFind c b with
        | Some t -> t :: aux (moveCoord c d) lst
        | None -> lst
    aux c []

(* Question 4.4 *)

let placeTile (c: coord, t: tile) (Board b) =
    match Map.tryFind c b with
    | Some _ -> None
    | None ->
        let upDown = collectTiles (Board b) (moveC
        let leftRight = collectTiles (Board b) (mo

        if validTiles upDown t && validTiles leftR
            let newMap = b.Add(c, t)
            Some(Board(newMap))
        else
            None

(* Question 4.5 *)

(* You may use *either* railroad-oriented programm
   You do not have to use both *)

(* Railroad-oriented programming *)
let ret = Some
let bind f =
    function
    | None   -> None
    | Some x -> f x
let (>>=) x f = bind f x

(* Computation expressions *)
```

```
    type opt<'a> = 'a option
    type OptBuilderClass() =
        member t.Return (x : 'a) : opt<'a> = ret x
        member t.ReturnFrom (x : opt<'a>) = x
        member t.Bind (o : opt<'a>, f : 'a -> opt<'b>)
    let opt = new OptBuilderClass()

    let placeTiles (ts: (coord * tile) list) (b: board
```

▼ Flyv's svar (med Railroad løsning)

```
module Question4

    (* Question 4.1 *)

    type color =
        | RED
        | GREEN
        | BLUE
        | PURPLE
        | ORANGE
        | YELLOW

    type shape =
        | SQUARE
        | CIRCLE
        | STAR
        | DIAMOND
        | CLUB
        | CROSS

    type tile = color * shape

    let matchColor =
        function
        | "red" -> RED
```

```
        | "green" -> GREEN
        | "blue" -> BLUE
        | "yellow" -> YELLOW
        | "purple" -> PURPLE
        | "orange" -> ORANGE
        | _ -> failwith "Not a valid color"

    let colorToString =
        function
        | RED -> "red"
        | BLUE -> "blue"
        | ORANGE -> "orange"
        | YELLOW -> "yellow"
        | PURPLE -> "purple"
        | GREEN -> "green"
    let matchShape =
        function
        | "diamond" -> DIAMOND
        | "square" -> SQUARE
        | "circle" -> CIRCLE
        | "club" -> CLUB
        | "cross" -> CROSS
        | "star" -> STAR
        | _ -> failwith "Not a valid shape"

    let shapeToString =
        function
        | DIAMOND -> "diamond"
        | CLUB -> "club"
        | STAR -> "star"
        | CIRCLE -> "circle"
        | SQUARE -> "square"
        | CROSS -> "cross"

    let mkTile color shape : tile = (matchColor color
    let tileToString (color, shape) =
```

```
        (colorToString color + " " + shapeToString sha

    (* Question 4.2 *)

    let validTiles tiles (newColor, newShape) =
        let tilesLength = List.length tiles
        let (colors, shapes) = List.fold (
                                    fun (colorAcc, shapeAc
                                        let newColorList =
                                        let newShapeList =
                                        (newColorList, new
                                    ) ([newColor], [newSha
        match List.length colors with
        | c when c = tilesLength+1 || c = 1 || c = 0 -
            //Colors correct
            match List.length shapes with
            | s when s = tilesLength+1 || s = 1 || s =
                //shapes correct
                true
            | _ -> false
        | _ -> false

    let validTiles2 (tiles: tile list) (tile:tile) =
        let tilesLength = List.length tiles
        match (List.distinctBy fst (tile::tiles) |> Li
        | c when c = tilesLength+1 || c = 1 || c = 0 -
            match (List.distinctBy snd (tile::tiles)
            | s when s = tilesLength+1 || s = 1 || s =
            | _ -> false
        | _ -> false;


    (* Question 4.3 optional *)

    type coord = Coord of int * int
    type board = Board of Map<coord, tile>
```

```
type direction = Left | Right | Up | Down

let moveCoord (Coord (x,y)) = function | Left -> (

let collectTiles (Board b) (c:coord) d =
    let rec aux c acc =
        match Map.tryFind c b with
        | None -> acc
        | Some tile -> aux (moveCoord c d) (tile:
    aux c []

(* Question 4.4 *)

let placeTile (Coord (x,y), tile:tile) (Board b)
    match Map.tryFind (Coord (x,y)) b with
    | Some _ -> None
    | None ->
        match validTiles2 (collectTiles (Board b)
        | false -> None
        | true ->
            match validTiles2 (collectTiles (Board
            | false -> None
            | true -> Some (Board (Map.add (Coord



(* Question 4.5 *)

(* You may use *either* railroad-oriented programm
   You do not have to use both *)

(* Railroad-oriented programming *)
let ret = Some
let bind f =
    function
    | None   -> None
```

```
        | Some x -> f x
    let (>>=) x f = bind f x

    (* Computation expressions *)
    type opt<'a> = 'a option
    type OptBuilderClass() =
        member t.Return (x : 'a) : opt<'a> = ret x
        member t.ReturnFrom (x : opt<'a>) = x
        member t.Bind (o : opt<'a>, f : 'a -> opt<'b>)
    let opt = new OptBuilderClass()

    let placeTiles l (b:board) =
        List.fold (
            fun (acc:board option) i ->
                acc >>= placeTile i
        ) (ret b) l
```

Exam 2019-05-16

https://itu.codejudge.net/func22/collection/3635/view

Question 1

```
module exam

// Question 1.1
type Peano =
    | O
    | S of Peano

let toInt p =
    let rec aux p n =
        match p with
        | O -> n
        | S p' -> aux p' (n + 1u)
    aux p 0u
```

```
let fromInt (i:uint32) =
    let rec aux i =
        match i with
        | 0u -> O
        | n -> S (aux (n - 1u))
    aux i

// Question 1.2
let rec add (p1:Peano) (p2:Peano) =
    match p1 with
    | O -> p2
    | S p1' -> add p1' (S p2)

let rec mult (p1:Peano) (p2:Peano) =
    match p1 with
    | O -> O
    | S p1' -> add p2 (mult p1' p2)

let rec pow (a:Peano) (b:Peano) =
    match b with
    | O -> S O
    | S n -> mult a (pow a n )

// Question 1.3
let tailAdd (p1:Peano) (p2:Peano) =
    let rec aux p acc =
        match p with
        | O -> acc
        | S p' -> aux p' (S acc)
    aux p1 p2

let tailMult (p1:Peano) (p2:Peano) =
    let rec aux p acc =
        match p with
        | O -> acc
        | S p' -> aux p' (tailAdd acc p2)
```

```
    aux p1 O

let tailPow (a:Peano) (b:Peano) =
    let rec aux p acc =
        match p with
        | O -> acc
        | S n -> aux n (tailMult acc a)
    aux b (S O)

// Question 1.4
let rec loop (f: 'a -> 'a) acc (p:Peano) =
    match p with
    | O -> acc
    | S p' -> loop f (f acc) p'

// Question 1.5
let loopAdd (a: Peano) (b: Peano) = loop S a b
let loopMult (a: Peano) (b: Peano) = loop (fun x -> loop
let loopPow (a: Peano) (b: Peano) = if b = O then (S O)
```

Question 2

Vores svar

```
let rec g xs =
    function
    | []    -> xs = []
    | y::ys ->
        match f y xs with
        | Some xs' -> g xs' ys
        | None     -> false

// Question 2.1
// What are the types of functions f and g
// f: 'a -> 'a list -> 'a list option when 'a: equalit
// g: 'a list -> 'a list -> bool when 'a: equality
```

```
// What do functions f and g do? Focus on what they do
// f: Takes 2 arguments, an element and a list. F remo
// g: Takes 2 arguments, and compares if they are equa

// What would be appropriate names for functions f and
// f: removeSingle
// g: equal

// Question 2.2

// The function f generates a warning during compilati
// Why does this happen, and where?
// It happens in f when matching on the list. It happe
// the two when statements complete each other, F# car
// Write a function f2 that does the same thing as f a

let rec f2 x =
    function
    | []                -> None
    | y::ys when x = y  -> Some ys
    | y::ys ->
        match f x ys with
        | Some ys' -> Some (y::ys')
        | None     -> None



let rec fOpt x lst =
    match lst with
    | [] -> None
    | y::ys when x = y -> Some ys
    | y::ys -> Option.map (fun ys' -> y :: ys') (fOpt

let rec gOpt xs ys =
    match ys with
    | [] -> xs = []
    | y :: ys -> Option.map (fun xs' -> gOpt xs' ys) (
```

```
printfn "%A" (gOpt [1; 2; 3; 4] [5; 4; 3; 2; 1])

// Question 2.4



(*
       g [1;2;3] [1;2]
    -> g [2;3] [2]
    -> g [3] []
    -> [3] = [] = false

*)

let fTail x xs =
    let rec aux x lst c =
        match lst with
        | [] -> None
        | y :: ys when x = y -> Some (c ys)
        | y :: ys -> aux x ys (fun r -> c (y :: r))
    aux x xs id
```

▼ **Jespers svar**

```
module CodeComprehension

    let rec f x =
        function
        | []                -> None
        | y::ys when x = y  -> Some ys
        | y::ys when x <> y ->
            match f x ys with
            | Some ys' -> Some (y::ys')
            | None     -> None

    let rec g xs =
```

```
        function
        | []    -> xs = []
        | y::ys ->
            match f y xs with
            | Some xs' -> g xs' ys
            | None     -> false

    (* Question 2.1 *)

    (*
    f has type x:'a -> _arg1:'a list -> 'a list optior
    g has type xs:'a list -> _arg1:'a list -> bool whe

    f x xs returns Some xs', where xs' is xs with the
           x exists in xs and None otherwise

    g xs ys returns true if xs is a permutation of ys

    f can be called removeFirstOccurence
    g can be called isPermutation

    (Another name for g could be containsSameElements,
     is anything regarding equality as the furction dc

    *)

    (* Question 2.2 *)

    (*
    The warning happens because F# does not realise tł

    (In the general case it is not possible to tell tł
     therefor F# does not even try.)
    *)

    let rec f2 x =
```

```
    function
    | []                -> None
    | y::ys when x = y  -> Some ys
    | y::ys ->
        match f2 x ys with
        | Some ys' -> Some (y::ys')
        | None     -> None

(* Question 2.3 *)

let cons x xs = x :: xs

let rec fOpt x =
    function
    | []                -> None
    | y::ys when x = y  -> Some ys
    | y::ys -> fOpt x ys |> Option.map (cons y)

(* Alternative solution *)
let rec fOpt2 x =
    function
    | []                -> None
    | y::ys when x = y  -> Some ys
    | y::ys -> Option.map (fun ys' -> y :: ys') (f

let rec gOpt xs =
    function
    | []    -> xs = []
    | y::ys -> fOpt y xs |> Option.map (gOpt ys)

(* Alternative solution *)
let rec gOpt2 xs =
    function
    | []    -> xs = []
    | y::ys -> Option.defaultValue false (Option.m
```

```
(* Question 2.4 *)

(*
g is tail recursive

Example

g [1; 2; 3] [3; 2; 1] ->
match f 3 [1; 2; 3] with
| Some xs' -> g xs' [2; 1]
| None     -> false ->
g [1; 2] [2; 1] ->
match f 2 [1; 2] with
| Some xs' -> g xs' [1]
| None     -> false ->
g [1] [1] ->
match f 1 [1] with
| Some xs' -> g xs' []
| None     -> false ->
g [] [] ->
true

We can see that g itself is tail recursive (even t
recursive occurrences of g appear completely by th

(The important thing for this assignment is that a
 to the exact same thing. If they do not then you
*)

(* Only implement the version of fTail or gTail th

let fTail x =
    let rec aux c =
        function
        | [] -> c None
        | y :: ys when x = y -> c (Some ys)
```

```
            | y :: ys -> aux (Option.map (cons y) >> c

        aux id


    (* Alternative solution *)
    let fTail2 x =
        let rec aux c =
            function
            | [] -> None
            | y :: ys when x = y -> Some (c ys)
            | y :: ys -> aux (fun ys' -> c (y :: ys'))

        aux id
```

▼ Question 3

▼ Question 4

▼ Assignments

▼ Assignment 1 - recursion

Assignment 1

▼ Green

```
// Exercise 1.1
let sqr x = x * x

// Exercise 1.2
let pow x n = System.Math.Pow(x, n)

// Exercise 1.3
let rec sum n =
    match n with
    | 0 -> 0
    | n -> n + sum(n - 1)
```

```
// Exercise 1.4
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | n -> fib(n - 1) + fib(n - 2)

// Exercise 1.5
let dup (s:string) = s + s

// Exercise 1.6
let rec dupn s n =
    match n with
    | 0 -> ""
    | n -> s + dupn s (n - 1)

// Exercise 1.7
let rec bin (n, k) =
    match (n, k) with
    | (_, 0) -> 1
    | (_, _) when n = k -> 1
    | (n, k) when n <> 0 && k <> 0 && n > k -> bin(n - 1
```

Yellow

```
// Exercise 1.8
let timediff (t1h, t1m) (t2h, t2m) =
    t2h * 60 + t2m - t1h * 60 - t1m

// Exercise 1.9
let minutes (h, m) =
    timediff (0, 0) (h, m)

// Exercise 1.10
```

```
let curry f x y = f (x,y)
let uncurry f (x, y) = f x y
```

Red

Scrabble related

Assignment 2 - recursion, lists, types

Assignment 2

Green

```
// Exercise 2.1
let rec downto1 n =
    if n <= 0 then []
    else n :: downto1 (n - 1)

let rec downto2 n =
    match n with
    | _ when n <= 0 -> []
    | n -> n :: downto2 (n - 1)

// Exercise 2.2
let rec removeOddIdx xs =
    match xs with
    | [] -> []
    | [x] -> [x]
    | x :: _ :: xs ->  x :: (removeOddIdx xs)

// Exercise 2.3
let rec combinePair xs =
    match xs with
    | [] -> []
    | [_] -> []
    | x :: y :: xs -> (x, y) :: (combinePair xs)

// Exercise 2.4
```

```
type complex = float * float
let mkComplex x y = complex (x, y)
let complexToPair complex = (fst complex, snd complex)

let (~-) ((a, b):complex) = (-a, -b)
let (~&) ((a, b):complex) = (a / (a**2. + b**2.), 0.0 -


let (|+|) ((a,b):complex) ((c,d):complex) = (a+c, b+d)
let (|*|) ((a,b):complex) ((c,d):complex) = (a*c - b*d,
let (|-|) (a: complex) (b: complex) = a |+| (-b)
let (|/|) (a: complex) (b: complex) = a |*| (&b)

// Exercise 2.5
let explode1 (s:string) = s.ToCharArray() |> List.ofArra

let rec explode2 (s:string) =
    match s with
    | ""-> []
    | s -> s.[0]::explode2 (s.[1..])

// Exercise 2.6
let implode (xs:char list) =
    List.foldBack (fun (x:char) (acc:string) -> (string

let implodeRev (xs:char list) =
    List.fold (fun (acc:string) (x:char) -> (string x) +

// Exercise 2.7
let toUpper s = s |> explode1 |> List.map (fun (c:char)
let toUpper2 = explode1 >> List.map (fun (c:char) -> Sys

// Exercise 2.8
let rec ack (m, n) =
    match m, n with
    | 0, _ -> n + 1
```

```
    | m, n when m > 0 && n = 0 -> ack (m - 1, 1)
    | m, n when m > 0 && n > 0 -> ack (m - 1, ack (m, n
    | _, _ -> 0
```

Yellow

```
// Exercise 2.9
let time f =
    let start = System.DateTime.Now
    let result = f ()
    let finish = System.DateTime.Now
    (result, finish - start)

let timeArg1 f a = time (fun () -> f a)

// Exercise 2.10
let rec downto3 f n e =
    match n with
    | _ when n <= 0 -> e
    | n -> downto3 f (n - 1) (f n e)

let fac a = downto3 (*) a 1
let range g n = downto3 (fun x acc -> (g x)::acc) n []
```

Red

Scrabble related

Assignment 3 - DSL, types

Assignment 3

Green

```
type aExp =
    | N of int
    | V of string
    | WL
```

```
    | PV of aExp
    | Add of aExp * aExp
    | Sub of aExp * aExp
    | Mul of aExp * aExp
    | Div of aExp * aExp

let (.+.) a b = Add (a, b)
let (.-.) a b = Sub (a, b)
let (.*.) a b = Mul (a, b)

let a1 = N 42;;
let a2 = N 4 .+. (N 5 .-. N 6)
let a3 = N 4 .*. N 2 .+. N 34
let a4 = (N 4 .+. N 2) .*. N 34
let a5 = N 4 .+. (N 2 .*. N 34)

let rec arithEvalSimple aExp =
    match aExp with
    | N n -> n
    | Add (a, b) -> arithEvalSimple a + arithEvalSimple
    | Sub (a, b) -> arithEvalSimple a - arithEvalSimple
    | Mul (a, b) -> arithEvalSimple a * arithEvalSimple
    | Div (a, b) -> arithEvalSimple a / arithEvalSimple

let a6 = V "x";;
let a7 = N 4 .+. (V "y" .-. V "z")

let rec arithEvalState aExp (s: Map<string, int>) =
    match aExp with
    | N n -> n
    | Add (a, b) -> (arithEvalState a s) + (arithEvalSta
    | Sub (a, b) -> (arithEvalState a s) - (arithEvalSta
    | Mul (a, b) -> (arithEvalState a s) * (arithEvalSta
    | Div (a, b) -> (arithEvalState a s) / (arithEvalSta
    | V v -> Map.tryFind v s |> Option.defaultValue 0
```

```
type word = (char * int ) list
let hello = ('H', 4)::('E', 1)::('L', 1)::('L', 1)::('O'

let arithSingleLetterScore = PV (V "_pos_") .+. (V "_acc
let arithDoubleLetterScore = ((N 2) .*. PV (V "_pos_"))
let arithTripleLetterScore = ((N 3) .*. PV (V "_pos_"))
let arithDoubleWordScore = N 2 .*. V "_acc_";;
let arithTripleWordScore = N 3 .*. V "_acc_";;

let rec arithEval aExp (w:word) (s: Map<string, int>) =
    match aExp with
    | N n -> n
    | Add (a, b) -> (arithEval a w s) + (arithEval b w s
    | Sub (a, b) -> (arithEval a w s) - (arithEval b w s
    | Mul (a, b) -> (arithEval a w s) * (arithEval b w s
    | Div (a, b) -> (arithEval a w s) / (arithEval b w s
    | V v -> Map.tryFind v s |> Option.defaultValue 0
    | WL -> List.length w
    | PV a -> snd w[arithEval a w s]

type cExp =
    | C of char
    | ToUpper of cExp
    | ToLower of cExp
    | CV of aExp

let rec charEval cExp (w:word) (s: Map<string, int>) =
    match cExp with
    | C c -> c
    | ToUpper a -> System.Char.ToUpper (charEval a w s)
    | ToLower a -> System.Char.ToLower (charEval a w s)
    | CV a -> fst w.[arithEval a w s]

type bExp =
    | TT
    | FF
```

```
    | AEq of aExp * aExp
    | ALt of aExp * aExp
    | Not of bExp
    | Conj of bExp * bExp
    | IsDigit of cExp
    | IsLetter of cExp
    | IsVowel of cExp

let (~~) b = Not b
let (.&&.) b1 b2 = Conj (b1, b2)
let (.||.) b1 b2 = ~~(~~b1 .&&. ~~b2) (* boolean disjunc
let (.=.) a b = AEq (a, b)
let (.<.) a b = ALt (a, b)
let (.<>.) a b = ~~(a .=. b) (* numeric inequality *)
let (.<=.) a b = a .<. b .||. ~~(a .<>. b) (* numeric le
let (.>=.) a b = ~~(a .<. b) (* numeric greater than or
let (.>.) a b = ~~(a .=. b) .&&. (a .>=. b) (* numeric g

let isVowel (c:char) =
    match c with
    | _ when "aeiouAEIOU".Contains c -> true
    | _ -> false

let rec boolEval bExp (w:word) (s: Map<string, int>) =
    match bExp with
    | TT -> true
    | FF -> false
    | AEq (a, b) -> (arithEval a w s) = (arithEval b w s
    | ALt (a, b) -> (arithEval a w s) < (arithEval b w s
    | Not b -> not (boolEval b w s)
    | Conj (b1, b2) -> (boolEval b1 w s) && (boolEval b2
    | IsDigit c -> System.Char.IsDigit (charEval c w s)
    | IsLetter c -> System.Char.IsLetter (charEval c w s
    | IsVowel c -> isVowel (charEval c w s)

let isConsonant cExp = Not (IsVowel cExp)
```

```
type stmnt =
    | Skip (* does nothing *)
    | Ass of string * aExp (* variable assignment *)
    | Seq of stmnt * stmnt (* sequential composition *)
    | ITE of bExp * stmnt * stmnt (* if-then-else statem
    | While of bExp * stmnt (* while statement *)
```

- Yellow

  Some yellow is also under green, but I didn't wanna do them all, since it is very much scrabble DSL related

- Red

  Scrabble related

Assignment 4 - MultiSet, Trie

Assignment 4

- Green
- Yellow
- Red

Assignment 5 - Tail recursion

Assignment 5

- Green

```
// Exercise 5.1
// tail recursion using accumulator
let sumA m n =
    let rec aux acc i =
        match i with
        | i when i = n -> acc + (m+i)
        | i -> aux (acc + (m + i)) (i + 1)
    aux 0 0

// Tail recursion using continuation
```

```
let sumC m n =
    let rec aux i c =
        match i with
        | i when i = n -> c (m+n)
        | i -> aux (i + 1) (fun r -> c ((m + i) + r))
    aux 0 id

// Exercise 5.2
// Tail recursion using accumulator
let lstLengthA lst =
    let rec aux acc lst =
        match lst with
        | [] -> acc
        | lst -> aux (acc + 1) (lst.Tail)
    aux 0 lst

// Tail recursion using continuation
let lstLengthC lst =
    let rec aux lst c =
        match lst with
        | [] -> c 0
        | lst -> aux (lst.Tail) (fun r -> c (r + 1))
    aux lst id

// Exercise 5.3
let foldback folder lst acc =
    let rec aux lst c =
        match lst with
        | [] -> c acc
        | x::xs-> aux xs (fun r -> c (folder x r))
    aux lst id

// Exercise 5.4
// Tail recursion using accumulator
let factA x =
    let rec aux acc =
```

```
        function
        | 0 -> acc
        | x -> aux (x * acc) (x - 1)
    aux 1 x

// Tail recursion using continuation
let factC x =
    let rec aux x c =
        match x with
        | 0 -> c 1
        | x -> aux (x-1) (fun r -> c (x * r))
    aux x id
```

▼ Yellow

```
// Exercise 5.5
// Tail recursion using accumulator
let fibA x =
    let rec aux x acc1 acc2 =
        match x with
        | 0 -> 0
        | 1 | 2 -> acc1 + acc2
        | x -> aux (x - 1) (acc2) (acc1 + acc2)
    aux x 0 1

// Tail recursion using continuation
let fibC x =
    let rec aux x c =
        match x with
        | 0 -> c 0
        | 1 | 2 -> c 1
        | x -> aux (x - 1) (fun r -> aux (x-2) (fun r' -
    aux x id
```

▼ Red

▼ Assignment 6

Assignment 6

▼ Green

▼ Yellow

▼ Red

▼ Assignment 7

Assignment 7

▼ Green

▼ Yellow

▼ Red

▼ Notes

▼ Nice helper functions / tricks / examples

String to char list

```
Seq.toList s
```

charToInt

```
let  charToInt c = int c - int '0'
```

char list to string

```
List.fold (fun acc elem -> elem + acc) "" list
```

gcd

```
let gcd x y =
    if y = 0 then x
```

```
    else gcd y (x % y)
```

Fast fibonacci

```
let memoisation f =
        let mutable m = Map.empty
        let aux x =
            match Map.tryFind x m with
            | Some y -> y
            | None ->
                m <- Map.add x (f x) m; f x
        aux

    let rec aux x : float=
        match x with
        | 0 | 1 -> 1.0
        | y -> fib (y - 1) + fib (y - 2)
    and fib = memoisation aux
```

parallel shit

```
(* Normal mapping function *)
let rec map f =
    function
    | [] -> []
    | x :: xs -> f x :: map f xs

(* Parallel mapping function *)
let parallelMap f lst =
    lst
    |> List.map (fun x -> async { // We need to create a li
        return f x  // "Asyncronously we want to return f x
    })
```

```
    |> Async.Parallel // Now we want to do these computati
    |> Async.RunSynchronously // Turns async 'T to a norma
    |> List.ofArray

(* My example of a parallel mapping function with "number 
let parallelMap2 f lst numberOfThreads =
    let threadSize = (List.length lst) / numberOfThreads
    lst
    |> List.chunkBySize threadSize // chunkBySize 2 [1;2;3
    |> List.map (fun l -> async {
        return List.map f l
    })
    |> Async.Parallel
    |> Async.RunSynchronously
    |> List.ofArray
    |> List.concat // we split the list with chunkBySize, 
```

Sequence and List comprehension

```
List comprehension
- [1..10]
- [for i in 1..10 do yield i]
- [for i in 1..10 -> i]

Sequence comprehension
- seq [1..10]
- let circleSphere x = seq {
    yield (circleArea x, sphereVolume x)
    yield! circleSphere (x+i)
  }
- seq { for i in 1..10 fo yield i }

let rec elSeq2 elem =
        seq {yield elem; yield! elSeq2 (nextElement elem)}
```

Tail recursion evaluation

```
// match case
Example

    g [1; 2; 3] [3; 2; 1] ->
    match f 3 [1; 2; 3] with
    | Some xs' -> g xs' [2; 1]
    | None     -> false ->
    g [1; 2] [2; 1] ->
    match f 2 [1; 2] with
    | Some xs' -> g xs' [1]
    | None     -> false ->
    g [1] [1] ->
    match f 1 [1] with
    | Some xs' -> g xs' []
    | None     -> false ->
    g [] [] ->
    true

// "normal"
    foo [1; 3] [2; 4; 5] -->
    1 :: foo [3] [2; 4; 5] -->
    1 :: 2 :: foo [3] [4; 5] -->
    1 :: 2 :: 3 :: foo [] [4; 5] -->
    1 :: 2 :: 3 :: [4; 5] -->
    [1; 2; 3; 4; 5]

bar [8; 7; 6; 5; 4; 3; 2; 1] -->
        foo (bar [8; 7; 6; 5])
            (bar [4; 3; 2; 1]) -->
        foo (foo (bar [8; 7])
                 (bar [6; 5]))
            (foo (bar [4; 3])
```

```
                  (bar [2; 1])) -->
         foo (foo (foo (bar [8]) (bar [7]))
                  (foo (bar [6]) (bar [5])))
             (foo (foo (bar [4]) (bar [3]))
                  (foo (bar [2]) (bar [1]))) -->
         foo (foo (foo [8] [7])
                  (foo [6] [5]))
             (foo (foo [4] [3])
                  (foo [2] [1])) -->
         foo (foo [7; 8] [5; 6])
             (foo [3; 4] [1; 2]) -->
         foo [5; 6; 7; 8] [1; 2; 3; 4] -->
         [1; 2; 3; 4; 5; 6; 7; 8]

// From book
   fact 4
-> 4 * fact(4-1)
-> 4 * fact 3
-> 4 * (3 * fact (3-1))
-> 4 * (3 * fact 2)
-> 4 * (3 * (2 * fact (2-1)))
-> 4 * (3 * (2 * (fact 1)))
-> 4 * (3 * (2 * (1 * fact (1-1)))
-> 4 * (3 * (2 * (1 * fact 0)))
-> 4 * (3 * (2 * (1 * 1)))
-> 4 * (3 * (2 * 1))
-> 4 * (3 * 2)
-> 4 * 6
-> 24
```

▼ Function signature

1 parameter

```
myFunc: int -> int
// Eksempel
let myFunc i = i*i
```

2 paremeters

```
myFunc2: float -> float -> float
// Eksempel
let myFunc2 a b = a+b
```

Tuple parameter

```
myFunc3: int * int -> int
// Eksempel
let myFunc3 (a, b) = a+b
```

Function parameter

```
// same as fn(5) + 2
let evalWith5ThenAdd2 fn = fn 5 + 2

// has type:
val evalWith5ThenAdd2 (int -> int) -> int

// define a function of type (int -> int)
let add1 x = x + 1

// test it works
evalWith5ThenAdd2 add1
val it : int = 8
```

strword & wordstr

```
// string to word in scrabble
let rec strwordrec input index word =
 if String.length(input) > index then
    strwordrec input (index+1) (add index (input.Chars(inde
 else word

let strword input = strwordrec input 0 (empty(char 0, 0))

let rec wordstrrec word index (str: string) =
 if uncurry (fun x y -> y) (word index) <> 0 then
    let char = (uncurry(fun x y -> x) (word index))
    wordstrrec word (index+1) (str + string char)
 else str

let wordstr input = wordstrrec input 0 ""
```

Sequences

```
let elSe = Seq.unfold (fun state -> Some (state, nextElemen

let rec elSeq2 elem =
        seq {yield elem; yield! elSeq2 (nextElement elem)}
```

Tuples

Example of working with tuples. Which time comes first, if they are AM and PM

```
// 3.1 triple
let t1 = (11, 59, "AM")
let t2 = (1, 15, "PM")

let time (a1, b1, c1) (a2, b2, c2) =
```

```
  match c1 with
  | c1 when c1 < c2 -> printfn "%A %A %s" a1 b1 c1
  | _ -> printfn "%A %A %s" a2 b2 c2
```

Records

Example of working with records. Which time comes first, if they are AM and PM

```
// 3.1 record
type Meridiem = AM | PM

type Time = {
  h: int;
  m: int;
  merid: Meridiem
}

let t1' = { h = 11; m = 59; merid = AM }
let t2' = { h = 1; m = 15; merid = PM }

let comesFirst { h = h1; m = m1; merid = t1 } { h = h2; m =
  match t1 with
  | t1 when t1 < t2 -> printfn "%A %A %A" h1 m1 t1
  | _ -> printfn "%A %A %A" h2 m2 t2
```

Tail recursion / Iterative functions

Recursive call is the last operation. F# will optimize the iterative functions so that they don't increase the stack.

Accumulators

Accumulators store the value that has been computed so far

```
let rec fact : int -> int =
    function
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Original factorial function

```
let rec factA (acc : int) : int -> int =
    function
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

factorial with an accumulator

“old” recursive vs. tail-recursive function. We keep the accumulator, multiply it continuously and do the recursive as the last operation.

```
factA 1 5   ⤳ factA (1 * 5) (5 - 1)    ⤳
factA 5 4   ⤳ factA (5 * 4) (4 - 1)    ⤳
factA 20 3  ⤳ factA (20 * 3) (3 - 1)   ⤳
factA 60 2  ⤳ factA (60 * 2) (2 - 1)   ⤳
factA 120 1 ⤳ factA (120 * 1) (1 - 1) ⤳
factA 120 0 ⤳ 120
```

Instead of

5 * (4 * (3 * (2 * (1 * 1))))

we compute

((((1 * 5) * 4) * 3) * 2) * 1

```
> rev [1..20000];;
Real: 00:00:17.506,
CPU:0:00:16.540,
GC gen0: 794, gen1: 0
val it : int list =
 [20000; 19999; 19998;…]
```

Naively reversing 20000 elements takes around 17 seconds and 794 garbage collections

```
> revA [1..20000];;
Real: 00:00:00.001,
CPU: 00:00:00.001,
GC gen0: 0, gen1: 0
val it : int list =
 [20000; 19999; 19998;…]
```

Reversing 20000 elements using an accumulator takes around 1 millisecond and no garbage collections

Tail recursion very efficient!

Accumulators are great when they work.

But they do not work all the time e.g. when

- We cannot reorder the way a function computes a result (e.g. the foldback function)
- We have multiple recursive calls that cannot be combined into one (e.g. tree traversal)

Continuations

```
let rec append xs ys =
    match xs with
    | []       -> ys
    | x :: xs -> x :: (append xs ys)
```

not a tail call

- An accumulator will not work (not directly at least).
- Instead, we use a continuation to get tail recursion
- A continuation is a function that s meant to be called after the recursive function

💡 **Idea**: make the recursive call, but remember that x still has to be added afterward

🥅 **Goal**: write a cursive function `appendC` that takes an additional argument

`c : ‘a list → ‘a list` such that

```
appendC xs ys c = c (append xs ys)
```

```
let rec append xs ys =
    match xs with
    | []      -> ys
    | x :: xs -> x :: (append xs ys)


let rec appendC xs ys c =
    match xs with
    | []       -> c ys
    | x :: xs  -> appendC xs  ys
                    (fun r -> c (x :: r))
```

We feed the value into the continuation.

```
    sum_cont [1,2] (fn x => x)
|->* sum_cont [2]   (fn s => (fn x => x) (1 + s))
|->* sum_cont []    (fn s' => (fn s => (fn x => x) (1 + s)) (2 + s'))
|->* (fn s' => (fn s => (fn x => x) (1 + s)) (2 + s')) 0
|->* (fn s => (fn x => x) (1 + s)) (2 + 0)
|->* (fn s => (fn x => x) (1 + s)) 2
|->* (fn x => x) (1 + 2)
|->* (fn x => x) 3
|->* 3
```

```
let sumC m n =
    let rec aux m c =
        match m with
        | m when m = n -> c m
        | m -> aux (m + 1) (fun r -> c (m + r))
    aux m id

sumC 0 3
aux 0 id
aux 1 (fun r' -> id (0 + r'))
aux 2 (fun r'' -> (fun r' -> id (0 + r')) (1 + r''))
aux 3 (fun r''' -> (fun r'' -> (fun r' -> id (0 + r')) (1 +
(fun r'' -> (fun r' -> id (0 + r')) (1 + r'')) 3
(fun r'' -> (fun r' -> id (0 + r')) (1 + 3))
(fun r' -> id (0 + r')) 4
(fun r' -> id (0 + 4))
id 4
4
```

Mutual recursion

Mutual recursion

Recursive type declaration:

```
type BinTree = Leaf
             | Node of BinTree * int * BinTree
```

Mutual recursive type declaration:

```
type RoseTree = Leaf
              | Node of int * Children
and Children  = RoseTree list
```

Two types that a are defined in terms of each other.

```
let rec sumRose (t : RoseTree) : int =
  match t with
  | Leaf -> 0
  | Node (n,ts) -> n + sumChildren ts
and sumChildren (ch : Children) : int =
  match ch with
  | [] -> 0
  | (t::ts) -> sumRose t + sumChildren ts
```

```
   sumRose (Node (4, [Leaf;Leaf]))
⤳ 4 + sumChildren [Leaf;Leaf]
⤳ 4 + (sumRose Leaf + sumChildren [Leaf])
⤳ 4 + (0 + (sumRose Leaf + sumChildren []))
⤳ 4 + (0 + (0 + 0))
```

⚠ We see the same issue as before. This will blow up our stack

Use continuations to solve this!

Continuations

```
let rec sumRoseC t (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (n,ts) -> sumChildrenC ts (fun s -> c (n + s))
and sumChC ch (c : int -> int) : int =
  match ch with
  | [] -> c 0
  | (t::ts) -> sumRoseC t
    (fun s -> sumChildrenC ts (fun s' -> c (s + s')))
```

```
  sumRoseC (Node (4, [Leaf])) id
⤳ sumChildrenC [Leaf] (fun s -> id (4 + s))
⤳ sumRoseC Leaf (fun s -> sumChildrenC []
     (fun s' ->  (fun s -> id (4 + s)) (s + s'))
⤳ sumChC [] (fun s' -> (fun s -> id (4 + s)) (0 + s'))
⤳ (fun s -> id (4 + s)) (0 + 0)  ⤳  id (4 + 0)  ⤳  4
```

Summary

- Recursive functions use the **stack** to remember where they are n the recursion
- This can lead to a **stack overflow**
- Solution: transform recursive functions into **iterative functions**, where recursive calls are always last
- Two approaches: **accumulator** (does not always work) & **continuation** (works always)
- This can also lead to **algorithmic improvements** (e.g. in rev)

Lecture Notes

Lecture notes

Scrabble Project

Inspiration

https://github.com/PhilipFlyvholm/EpicScrabbleProject https://github.com/4lgn/scrabble-bot/blob/master/ScrabbleBot/Scrabble.fs

https://github.com/FrederikRothe/ScrabbleProject

Master plan

> ✅ When a move is a successful move received, whether we played it or someone else
>
> - For each letter played, store the letter and the coordinates in a hashmap in the state

- For each letter
  - Move down procedure:
    - Check if there if there is a letter above, if there is break
    - move down until empty we hit an empty space, return the word found so far
    - Store the word in map with the transform (pos + dir) as value and word as key
    - For each word, attempt to extend the word by searching in the trie with word found so far as start and letters on hand as continuation.
    - Store each word extension found as a possible move in a map, with the number of points as the key, value as the move
  - Move right procedure:
    - same as move down procedure

  - Sort the move map
  - Play the move with the greatest number of points (ignore special squares)