IT UNIVERSITY OF COPENHAGEN

Functional Programming

Patrick Bahr

Mutual Recursion & Sequences

Based on original slides by Michael R. Hansen

Last week

- The memory model of F#
- Tail-recursive functions
  (aka. iterative functions)
  - Tail-recursion with accumulators
  - Tail-recursion with continuations

This week

- Mutual Recursion
- Sequences
  - Lazy evaluation
  - Sequence expressions

Mutual recursion

Recursive function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Mutual recursion

Recursive function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Mutual recursive function:

```
let rec isEven n = if n = 0 then true
                   else isOdd (n - 1)

and     isOdd  n = if n = 0 then false
                   else isEven (n - 1)
```

Mutual recursion

Recursive function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Mutual recursive function:

```
let rec isEven n = if n = 0 then true
                   else isOdd (n - 1)

and     isOdd  n = if n = 0 then false
                   else isEven (n - 1)
```

Mutual recursion

Recursive function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Mutual recursive function:

```
let rec isEven n = if n = 0 then true
                   else isOdd (n - 1)

and     isOdd  n = if n = 0 then false
                   else isEven (n - 1)
```

Mutual recursion

Recursive function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Mutual recursive function:

```
let rec isEven n = if n = 0 then true
                   else isOdd (n - 1)

and     isOdd  n = if n = 0 then false
                   else isEven (n - 1)
```

Continuations

Recall: Tree traversal

```
type BinTree =
    | Leaf
    | Node of BinTree * int * BinTree

let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n
```

Recall: Tree traversal

```
type BinTree =
    | Leaf
    | Node of BinTree * int * BinTree

let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let t = genTree 1000000
printfn "%d" (sum t)
```

generates a tree of height 1,000,000

Recall: Tree traversal

```
type BinTree =
    | Leaf
    | Node of BinTree * int * BinTree

let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let t = genTree 1000000
printfn "%d" (sum t)
```

generates a tree of height 1,000,000

⇒ Stack overflow

Let's Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let rec sumA (t : BinTree) (acc : int) : int =
    match t with
    | Leaf -> acc
    | Node (l,n,r) -> sumA r (sumA l (n + acc))
```

Let’s Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let rec sumA (t : BinTree) (acc : int) : int =
    match t with
    | Leaf -> acc
    | Node (l,n,r) -> sumA r (sumA l (n + acc))
```

not a tail call

Let’s Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let rec sumA (t : BinTree) (acc : int) : int =
    match t with
    | Leaf -> acc
    | Node (l,n,r) -> sumA r (sumA l (n + acc))
```

not a tail call

We need a way to say “compute sum of `l` and afterwards **continue** with `r`” while only using one recursive call

Let’s Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
     | Leaf -> 0
     | Node (l,n,r) -> sum l + sum r + n
```

Goal: write a recursive function

```
sumC : BinTree -> (int -> int) -> int
```

such that

```
sumC t c = c (sum t)
```

Hence: `sum t = sumC t id`

Let’s Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n
```

Goal: write a recursive function

```
sumC : BinTree -> (int -> int) -> int
```

such that

```
sumC t c = c (sum t)
```

Hence: `sum t = sumC t id`

sum with a continuation

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n


let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
sumC (Node (Leaf, 4, Leaf)) id
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
  sumC (Node (Leaf, 4, Leaf)) id
⇝ sumC Leaf (fun vl -> sumC Leaf
                (fun vr -> id (vl + vr + 4)))
```

IT UNIVERSITY OF COPENHAGEN

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
               sumC r (fun vr -> c (vl + vr + n)))
```

```
   sumC (Node (Leaf, 4, Leaf)) id
⤳ sumC Leaf (fun vl -> sumC Leaf
                (fun vr -> id (vl + vr + 4)))
⤳ sumC Leaf (fun vr -> id (0 + vr + 4))
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
  sumC (Node (Leaf, 4, Leaf)) id
⤳ sumC Leaf (fun vl -> sumC Leaf
                (fun vr -> id (vl + vr + 4)))
⤳ sumC Leaf (fun vr -> id (0 + vr + 4))
⤳ id (0 + 0 + 4)
```

sum with a continuation

IT UNIVERSITY OF COPENHAGEN

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
  sumC (Node (Leaf, 4, Leaf)) id
⇝ sumC Leaf (fun vl -> sumC Leaf
                 (fun vr -> id (vl + vr + 4)))
⇝ sumC Leaf (fun vr -> id (0 + vr + 4))
⇝ id (0 + 0 + 4)    ⇝ 4
```

Part I

Mutual Recursion

Mutual recursion

Recursive type declaration:

```
type BinTree = Leaf
             | Node of BinTree * int * BinTree
```

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.

Mutual recursion

To traverse mutual recursive types we need **mutually recursive functions**, i.e. functions that call 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
```

Mutual recursion

To traverse mutual recursive types we need **mutually recursive functions**, i.e. functions that call 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
```

Mutual recursion

```
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
```

Mutual recursion

```
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]))
```

Mutual recursion

```
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]
```

IT UNIVERSITY OF COPENHAGEN

Mutual recursion

```
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])
```

Mutual recursion

```
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 []))
```

Mutual recursion

```
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))
```

Mutual recursion

```
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)) ⤳ 4
```

Tail recursion with continuations

```
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
```

Let's try to make these mutual recursive functions **tail recursive**!

Continuations

```
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
```

```
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 sumChildrenC ch (c : int -> int) : int =
  match ch with
  | [] -> c 0
  | (t::ts) -> sumRoseC t
    (fun s -> sumChildrenC ts (fun s' -> c (s + s')))
```

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 sumChildrenC ch (c : int -> int) : int =
  match ch with
  | [] -> c 0
  | (t::ts) -> sumRoseC t
    (fun s -> sumChildrenC ts (fun s' -> c (s + s')))
```

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 sumChildrenC ch (c : int -> int) : int =
  match ch with
  | [] -> c 0
  | (t::ts) -> sumRoseC t
    (fun s -> sumChildrenC ts (fun s' -> c (s + s')))
```

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 sumChildrenC 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
```

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 sumChildrenC 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))
```

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 sumChildrenC 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'))
```

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 sumChildrenC 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'))
⤳ sumChildrenC [] (fun s' -> (fun s -> id (4 + s)) (0 + s'))
```

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 sumChildrenC 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'))
⤳ sumChildrenC [] (fun s' -> (fun s -> id (4 + s)) (0 + s'))
⤳ (fun s -> id (4 + s)) (0 + 0)
```

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 sumChildrenC 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'))
⤳ sumChildrenC [] (fun s' -> (fun s -> id (4 + s)) (0 + s'))
⤳ (fun s -> id (4 + s)) (0 + 0)  ⤳  id (4 + 0)
```

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 sumChildrenC 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'))
⇝ sumChildrenC [] (fun s' -> (fun s -> id (4 + s)) (0 + s'))
⇝ (fun s -> id (4 + s)) (0 + 0)  ⇝  id (4 + 0)  ⇝  4
```

Polymorphic Continuations

Let's write a function to increment the numbers inside a tree:

Polymorphic Continuations

Let's write a function to increment the numbers inside a tree:

```
let rec incRose (t : RoseTree) : RoseTree =
  match t with
  | Leaf -> Leaf
  | Node (n, ts) -> Node (n+1,incCh ts)
and incCh (ch : Children) : Children =
  match ch with
  | [] -> []
  | t :: ts -> incRose t :: incCh ts
```

Polymorphic Continuations

Let's write a function to increment the numbers inside a tree:

```
let rec incRose (t : RoseTree) : RoseTree =
  match t with
  | Leaf -> Leaf
  | Node (n, ts) -> Node (n+1,incCh ts)
and incCh (ch : Children) : Children =
  match ch with
  | [] -> []
  | t :: ts -> incRose t :: incCh ts
```

Let's try to come up with a tail-recursive version.

Polymorphic Continuations

```
let rec incRoseC t (c : RoseTree -> 'a) : 'a =
  match t with
  | Leaf -> c Leaf
  | Node (n, ts) ->
    incChC ts (fun r -> c (Node (n+1, r)))
and incChC ch (c : Children -> 'a) : 'a =
  match ch with
  | [] -> c []
  | t :: ts ->
      incRoseC t (fun t' ->
        incChC ts (fun ts' -> c (t' :: ts')))
```

Continuations need to be polymorphic here!

Questions?

Part II

Sequences

Sequences

- We are already very familiar with **lists** (type `list<'a>`)
- **Sequences** (type `seq<'a>`) behave a lot like lists, but
  - sequences can be **lazy**
    (i.e. their elements may be computed on demand)
  - sequences can be **infinite**
    (and that can be useful!)
- We'll see shortly why lazy and/or infinite sequences might be useful

Constructing sequences

Sequences can be created inline similarly to lists

```
> seq [1; 2; 3];;
val it : seq<int> = [1; 2; 3]
```

Constructing sequences

We can use functions to create *finite* sequences:

```
Seq.init : int -> (int -> 'a) -> seq<'a>
Seq.init n f = seq[f 0; f 1; … ; f (n-1)]
```

Constructing sequences

We can use functions to create *finite* sequences:

```
Seq.init : int -> (int -> 'a) -> seq<'a>
Seq.init n f = seq[f 0; f 1; … ; f (n-1)]
```

```
> Seq.init 4 (fun n -> n*n);;
val it : seq<int> = seq [0; 1; 4; 9]
```

Constructing sequences

… and *infinite* sequences:

```
Seq.initInfinite : (int -> 'a) -> seq<'a>
Seq.initInfinite f = seq[f 0; f 1; …]
```

Constructing sequences

… and *infinite* sequences:

```
Seq.initInfinite : (int -> 'a) -> seq<'a>
Seq.initInfinite f = seq[f 0; f 1; …]
```

```
> Seq.initInfinite (fun n -> n*n);;
val it : seq<int> = seq [0; 1; 4; 9; ...]
```

Constructing sequences

... and *infinite* sequences:

```
Seq.initInfinite : (int -> 'a) -> seq<'a>
Seq.initInfinite f = seq[f 0; f 1; …]
```

```
> Seq.initInfinite (fun n -> n*n);;
val it : seq<int> = seq [0; 1; 4; 9; ...]
```

How can we work with infinite objects??

Lazy Evaluation

Lazy evaluation (or *delayed evaluation*)	=	Computation is delayed until its result is *needed*.

Lazy Evaluation

Lazy evaluation (or *delayed evaluation*)	=	Computation is delayed until its result is *needed*.

By default F# does *not* use lazy evaluation.

Lazy Evaluation

Lazy evaluation (or *delayed evaluation*) = Computation is delayed until its result is *needed*.

By default F# does *not* use lazy evaluation.

Example:

```
> let f x =
      let y = (x * x, x + x)
      fst y;;
> f 5;;
val it: int = 25
```

Both x * x and x + x are evaluated, even though x + x is not needed.

Lazy Evaluation

Lazy evaluation (or *delayed evaluation*) = Computation is delayed until its result is *needed*.

By default F# does *not* use lazy evaluation.

Example:

```
> let f x =
      let y = (x * x, x + x)
      fst y;;
> f 5;;
val it: int = 25
```

Both x * x and x + x are evaluated, even though x + x is not needed.

Lazy Evaluation

Lazy evaluation
(or *delayed evaluation*) = Computation is delayed until its result is *needed*.

Lazy Evaluation

Lazy evaluation (or *delayed evaluation*) = Computation is delayed until its result is *needed*.

- Some languages are lazy by default (e.g. Haskell)
- We can *occasionally* use lazy evaluation in F#
- But: Lazy evaluation must be used with care since delaying evaluation is costly (why?)
- Elements of infinite sequences are lazily evaluated.

Delayed computations

The computation of the value of *e* can be delayed by "packing" it into a function (called a closure):

```
fun () -> e
```

Delayed computations

The computation of the value of *e* can be delayed by "packing" it into a function (called a closure):

```
fun () -> e
```

Example

```
> fun () -> 3 + 4;;
val it : unit -> int

> it ();;
val it : int = 7
```

The expression **3 + 4** is not evaluated until the function is called

Example

We can make it visible when computations are performed by the use of side effects:

```
let idWithPrint (i : int) : int =
  printfn "%d" i
  i
```

The above function takes an integer, prints it out, and finally returns it again

Examples

```
> idWithPrint 3;;
3
val it : int = 3

> (idWithPrint 3) + 1;;
3
val it : int = 4
```

Example

We can make it visible when computations are performed by the use of side effects:

```
let idWithPrint (i : int) : int =
  printfn "%d" i
  i
```

The above function takes an integer, prints it out, and finally returns it again

Examples

```
> fun () ->
    (idWithPrint 3) +
      (idWithPrint 4);;
val it : unit -> int
```

```
> it();;
3
4
val it : int = 7
```

Example

We can make it visible when computations are performed by the use of side effects:

Note: Nothing is printed yet!

```
let idWithPrint (i : int) : int
  printfn "%d" i
  i
```

The above function takes an integer, prints it out, and finally returns it again

Examples

```
> fun () ->
    (idWithPrint 3) +
      (idWithPrint 4);;
val it : unit -> int
```

```
> it();;
3
4
val it : int = 7
```

Example

We can make it visible when computations are performed by the use of side effects:

Note: Nothing is printed yet!

```
let idWithPrint (i : int) : int
  printfn "%d" i
  i
```

The
an i
and

Once we apply the function, the expression in it is evaluated

Examples

```
> fun () ->
    (idWithPrint 3) +
      (idWithPrint 4);;
val it : unit -> int

> it();;
3
4
val it : int = 7
```

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

The following will evaluate to the value 42

> abc 42;;

It will also print the strings "A", "B", and "C". But in which order?

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

```
> abc 42;;
```

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

```
> abc 42;;
A
```

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

```
> abc 42;;
A
B
```

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

```
> abc 42;;
A
B
C
```

Quiz

```
let abc x =
  let foo =
      printfn "A"
      x
  let bar = fun () ->
      printfn "B"
      foo
  let foobar () =
      let r = bar ()
      printfn "C"
      r
  foobar ()
```

```
> abc 42;;
A
B
C
val it : int = 42
```

Sequences

A sequence is a (possibly infinite) ordered collection of values.

```
> let nat = Seq.initInfinite (fun x -> x);;     // seq [0; 1; 2; 3; …]
val nat : seq<int>

> let pos = Seq.initInfinite (fun x -> x + 1);; // seq [1; 2; 3; 4; …]
val pos : seq<int>
```

```
> Seq.item 4 nat;;
val it : int = 4
```

```
> Seq.item 4 pos;;
val it : int = 5
```

Sequences

Items from sequences are computed on demand

```
> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>                              // seq [0; 1; 2; 3; …]

> Seq.item 4 nat;;
4
val it : int = 4

> Seq.item 5 nat;;
5
val it : int = 5

> Seq.item 4 nat;;
4
val it : int = 4
```

Sequences

Items from sequences are computed on demand

```
> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>                        // seq [0; 1; 2; 3; …]

> Seq.item 4 nat;;
4
val it : int = 4

> Seq.item 5 nat;;
5
val it : int = 5

> Seq.item 4 nat;;
4
val it : int = 4
```

```
let idWithPrint (i : int) : int =
  printfn "%d" i
  i
```

Sequences

Items from sequences are computed on demand

```
> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>                          // seq [0; 1; 2; 3; …]
```

```
let idWithPrint (i : int) : int =
  printfn "%d" i
  i
```

```
> Seq.item 4 nat;;
4
val it : int = 4

> Seq.item 5 nat;;
5
val it : int = 5

> Seq.item 4 nat;;
4
val it : int = 4
```

We have to **recompute** the element of the sequence

Caching

Recomputation can be avoided by using a *cache*

```
Seq.cache : seq<'a> -> seq<'a>

> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>

> let natCache = Seq.cache nat;;
val natCache : seq<int>
```

Caching

Recomputation can be avoided by using a *cache*

```
Seq.cache : seq<'a> -> seq<'a>
```

```
> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>

> let natCache = Seq.cache nat;;
val natCache : seq<int>
```

Example

```
> Seq.item 2 natCache;;
0
1
2
val it : int = 2
```

Caching

Recomputation can be avoided by using a *cache*

```
Seq.cache : seq<'a> -> seq<'a>
```

```
> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>

> let natCache = Seq.cache nat;;
val natCache : seq<int>
```

Example

```
> Seq.item 2 natCache;;
0
1
2
val it : int = 2

> Seq.item 2 natCache;;
val it : int = 2
```

Caching

Recomputation can be avoided by using a *cache*

```
Seq.cache : seq<'a> -> seq<'a>
```

```
> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>

> let natCache = Seq.cache nat;;
val natCache : seq<int>
```

Example

```
> Seq.item 2 natCache;;
0
1
2
val it : int = 2

> Seq.item 2 natCache;;
val it : int = 2

> Seq.item 4 natCache;;
3
4
val it : int = 4
```

Caching

Recomputation can be avoided by using a *cache*

```
Seq.cache : seq<'a> -> seq<'a>
```

```
> let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>

> let natCache = Seq.cache nat;;
val natCache : seq<int>
```

- a cached sequence has first n elements cached
- when looking up element i < n, then cached value is returned (no computation required)
- otherwise (i ≥ n) the elements n, … , i are computed and stored in the cache

Example

```
> Seq.item 2 natCache;;
0
1
2
val it : int = 2

> Seq.item 2 natCache;;
val it : int = 2

> Seq.item 4 natCache;;
3
4
val it : int = 4
```

Filtering

A sequence of even natural numbers is obtained by filtering

```
> let even = Seq.filter (fun n -> n%2=0) nat;;
val even : seq<int>

> Seq.toList(Seq.take 4 even);;
0
1
2
3
4
5
6
val it : int list = [0; 2; 4; 6]
```

Filtering

A sequence of even natural numbers is obtained by filtering

```
> let even = Seq.filter (fun n -> n%2=0) nat;;
val even : seq<int>
```

```
> Seq.toList(Seq.take 4 even);;
0
1
2
3
4
5
6
val it : int list = [0; 2; 4; 6]
```

Calculating the first four even numbers requires computing the first seven natural numbers

Filtering

A sequence of even natural numbers is obtained by filtering

```
> let even = Seq.filter (fun n -> n%2=0) nat;;
val even : seq<int>
```

```
> Seq.toList(Seq.take 4 even);;
0
1
2
3
4
5
6
val it : int list = [0; 2; 4; 6]
```

Calculating the first four even numbers requires computing the first seven natural numbers

By using an infinite sequence, we don't have to worry about how many natural numbers we need to check!

Delay

Let's try to define `seq [0; 1; 2; 3; …]` by recursion:

```
let rec from n =
    Seq.append (seq [n]) (from (n+1))
```

Delay

Let's try to define `seq [0; 1; 2; 3; …]` by recursion:

```
let rec from n =
    Seq.append (seq [n]) (from (n+1))
```

Idea

`from n = seq [n; n+1;…]`

Delay

Let's try to define `seq [0; 1; 2; 3; …]` by recursion:

```
let rec from n =
    Seq.append (seq [n]) (from (n+1))
```

`from 0` will loop forever!

```
> from 0;;
  Stack overflow
```

Delay

Let's try to define `seq [0; 1; 2; 3; …]` by recursion:

```
let rec from n =
    Seq.append (seq [n]) (from (n+1))
```

`from 0` will loop forever!

```
> from 0;;
  Stack overflow
```

We have to delay the recursive call to `from`

```
Seq.delay : (unit -> seq<'a>) -> seq<'a>
```

```
let rec from n =
    Seq.append (seq [n])
      (Seq.delay (fun () -> from (n+1)))
```

Delay

Let's try to define `seq [0; 1; 2; 3; …]` by recursion:

```
let rec from n =
    Seq.append (seq [n]) (from (n+1))
```

`from 0` will loop forever!

```
> from 0;;
  Stack overflow
```

We have to delay the recursive call to `from`

```
Seq.delay : (unit -> seq<'a>) -> seq<'a>
```

```
let rec from n =
    Seq.append (seq [n])
      (Seq.delay (fun () -> from (n+1)))
```

```
> let nat = from 0;;
val nat : seq<int>

> Seq.item 4 nat;;
val it : int = 4
```

Part III

Sequence Expressions

Sieve of Eratosthenes

Computing the sequence of prime numbers using the following simple procedure:

- Start with the sequence 2, 3, 4, 5, 6, …

Sieve of Eratosthenes

Computing the sequence of prime numbers using the following simple procedure:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …

Sieve of Eratosthenes

Computing the sequence of prime numbers using the following simple procedure:

$\boxed{2,}$ 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence

Sieve of Eratosthenes

Computing the sequence of prime numbers
using the following simple procedure:

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence

Sieve of Eratosthenes

Computing the sequence of prime numbers
using the following simple procedure:

$\boxed{2}$, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence
- Next sequence is 3, 5, 7, 9, 11, …

Sieve of Eratosthenes

Computing the sequence of prime numbers using the following simple procedure:

2, $\boxed{3}$, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence
- Next sequence is 3, 5, 7, 9, 11, …
  select the head (3) and remove multiples of 3 from the sequence

Sieve of Eratosthenes

Computing the sequence of prime numbers using the following simple procedure:

2, $\boxed{3}$, $\cancel{4}$, 5, $\cancel{6}$, 7, $\cancel{8}$, $\cancel{9}$, $\cancel{10}$, 11, $\cancel{12}$, 13, $\cancel{14}$, $\cancel{15}$, $\cancel{16}$, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence
- Next sequence is 3, 5, 7, 9, 11, …
  select the head (3) and remove multiples of 3 from the sequence

Sieve of Eratosthenes

Computing the sequence of prime numbers
using the following simple procedure:

2, $\boxed{3}$, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence
- Next sequence is 3, 5, 7, 9, 11, …
  select the head (3) and remove multiples of 3 from the sequence
- Next sequence is 5, 7, 11, 13, 17

Sieve of Eratosthenes

Computing the sequence of prime numbers using the following simple procedure:

2, 3, ~~4~~, $\boxed{5}$, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence
- Next sequence is 3, 5, 7, 9, 11, …
  select the head (3) and remove multiples of 3 from the sequence
- Next sequence is 5, 7, 11, 13, 17
  select the head (5) and remove multiples of 5 from the sequence

Sieve of Eratosthenes

Computing the sequence of prime numbers using the following simple procedure:

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, …

- Start with the sequence 2, 3, 4, 5, 6, …
  select the head (2) and remove multiples of 2 from the sequence
- Next sequence is 3, 5, 7, 9, 11, …
  select the head (3) and remove multiples of 3 from the sequence
- Next sequence is 5, 7, 11, 13, 17
  select the head (5) and remove multiples of 5 from the sequence
- …

Sieve of Eratosthenes

Remove multiples of **a** from sequence **sq**

```
let sift a sq = Seq.filter (fun n -> n % a <> 0) sq
```

Sieve of Eratosthenes

Remove multiples of **a** from sequence **sq**

```
let sift a sq = Seq.filter (fun n -> n % a <> 0) sq
```

Select the head and remove multiples of the head from the sequence

```
let rec sieve sq =
    Seq.delay (fun () ->
        let p = Seq.item 0 sq
        Seq.append
            (Seq.singleton p)
            (sieve(sift p (Seq.skip 1 sq))))
```

Sieve of Eratosthenes

Remove multiples of **a** from sequence **sq**

```
let sift a sq = Seq.filter (fun n -> n % a <> 0) sq
```

Select the head and remove multiples of the head from the sequence

```
let rec sieve sq =
    Seq.delay (fun () ->
        let p = Seq.item 0 sq
        Seq.append
            (Seq.singleton p)
            (sieve(sift p (Seq.skip 1 sq))))
```

Seq.delay is needed to avoid inifinite recursion

Putting it all together

```
let numFrom2 = Seq.initInfinite (fun n -> n+2)    // seq [2; 3; 4; …]
let primes   = sieve numFrom2                      // seq [2; 3; 5; …]
```

Putting it all together

```
let numFrom2 = Seq.initInfinite (fun n -> n+2)    // seq [2; 3; 4; …]
let primes   = sieve numFrom2                     // seq [2; 3; 5; …]

let nthPrime n = Seq.item n primes                // n-th prime
```

Putting it all together

```
let numFrom2 = Seq.initInfinite (fun n -> n+2)   // seq [2; 3; 4; …]
let primes   = sieve numFrom2                    // seq [2; 3; 5; …]

let nthPrime n = Seq.item n primes               // n-th prime
```

```
> nthPrime 1000;;
 Real: 00:00:07.200, CPU: 00:00:07.209, GC gen0: 272, gen1: 3
 val it : int = 7927

> nthPrime 1001;;
 Real: 00:00:07.021, CPU: 00:00:07.081, GC gen0: 272, gen1: 3
 Val it : int = 7933
```

Caching the sequence of primes

Recomputation can be avoided by using a cache

```
let primesCached =
    Seq.cache primes

let nthPrime' n =
    Seq.item n primesCached
```

Caching the sequence of primes

Recomputation can be avoided by using a cache

```
let primesCached =
    Seq.cache primes

let nthPrime' n =
    Seq.item n primesCached
```

```
> nthPrime' 1000;;
 Real: 00:00:07.023, CPU: 00:00:07.056, GC gen0: 272, gen1: 2
 val it : int = 7927

> nthPrime' 1001;;
 Real: 00:00:00.021, CPU: 00:00:00.023, GC gen0: 0, gen1: 0
 Val it : int = 7933
```

Sieve of Eratosthenes

We can use sequence expressions to write sieve

```
let rec sieve sq =
    seq { let p = Seq.item 0 sq
          yield p
          yield! sieve (sift p (Seq.skip 1 sq)) }
```

- Implicitly lazy construction; **Seq.delay** is not needed
- **yield x** inserts the element **x**
- **yield! sq** inserts the sequence **sq**

Sieve of Eratosthenes

```
let rec sieve sq =
    seq { let p = Seq.item 0 sq
          yield p
          yield! sieve (sift p (Seq.skip 1 sq)) }
let sift a sq =
    seq { for n in sq do
            if n % a <> 0 then yield n }
```

- **`for p in sq do`** iterates over sequence **sq**
- **`if b then sq`** only includes **sq** if **b** is true

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}

val it : seq<int> = seq [1; 4; 9; 16; …]
```

- Sequence expressions are a special case of **computation expressions**.
- We'll learn about these next week!
- This is just a preview.

A computation expression is defined like this:

```
type MySeqBuilder() =
  member this.Yield x = …
  …
let seq = MySeqBuilder()
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

The above syntax is translated as follows:

```
T(yield v)               = Yield v

T(yield! e)              = YieldFrom e

T(for x in e do ce)      = For(e, fun x -> T(ce))
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

The above syntax is translated as follows:

T is the function that does the translation.

```
T(yield v)              = Yield v

T(yield! e)             = YieldFrom e

T(for x in e do ce)     = For(e, fun x -> T(ce))
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

In addition, computation expressions for sequences use the function `Delay`

```
T(seq {ce})         = Delay (fun () -> T (ce))
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

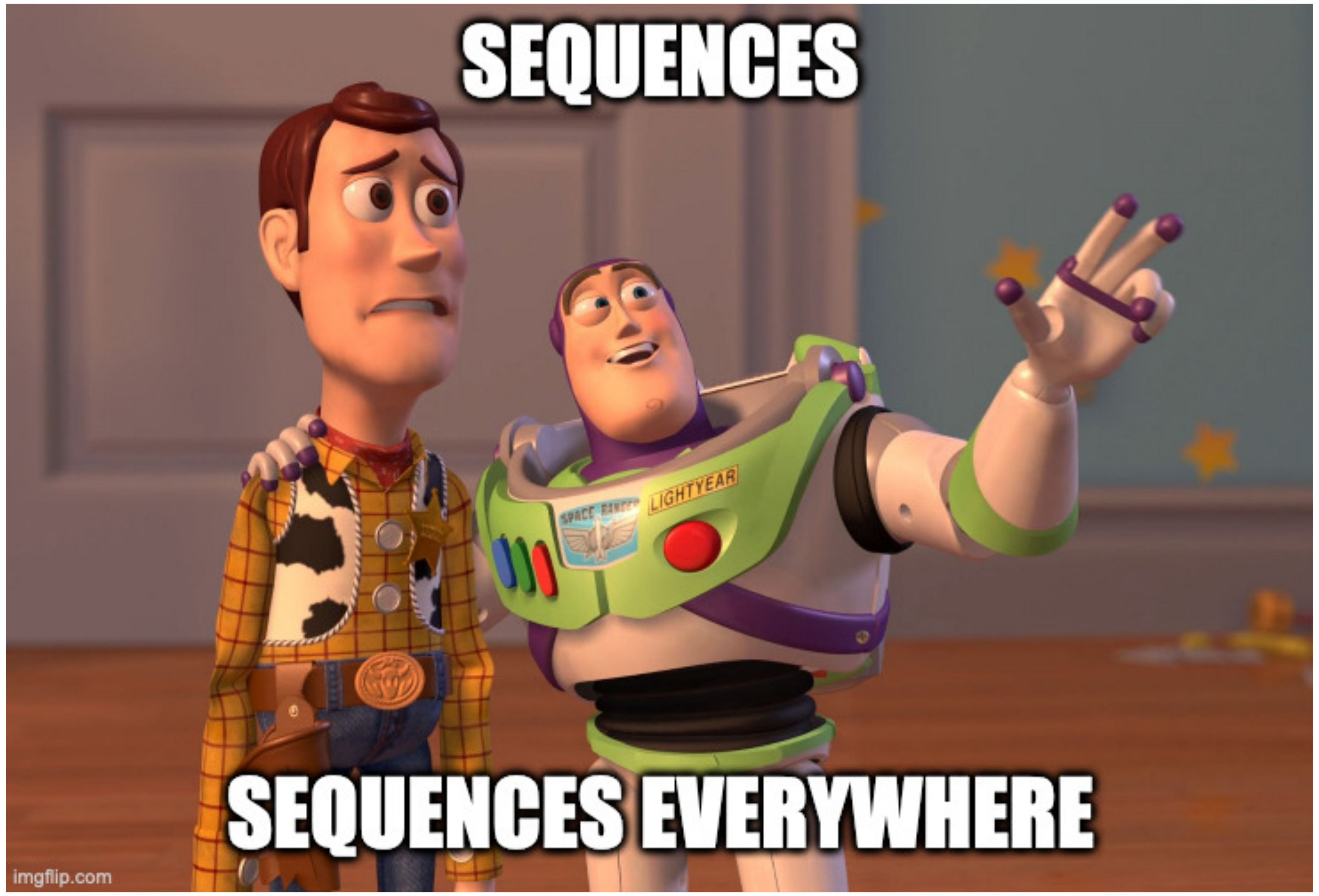
In addition, computation expressions for sequences use the function `Delay`

```
T(seq {ce})        = Delay (fun () -> T (ce))
```

This makes sure that sequences defined by `seq {…}` are delayed by default (⇒ can be used in recursive functions)

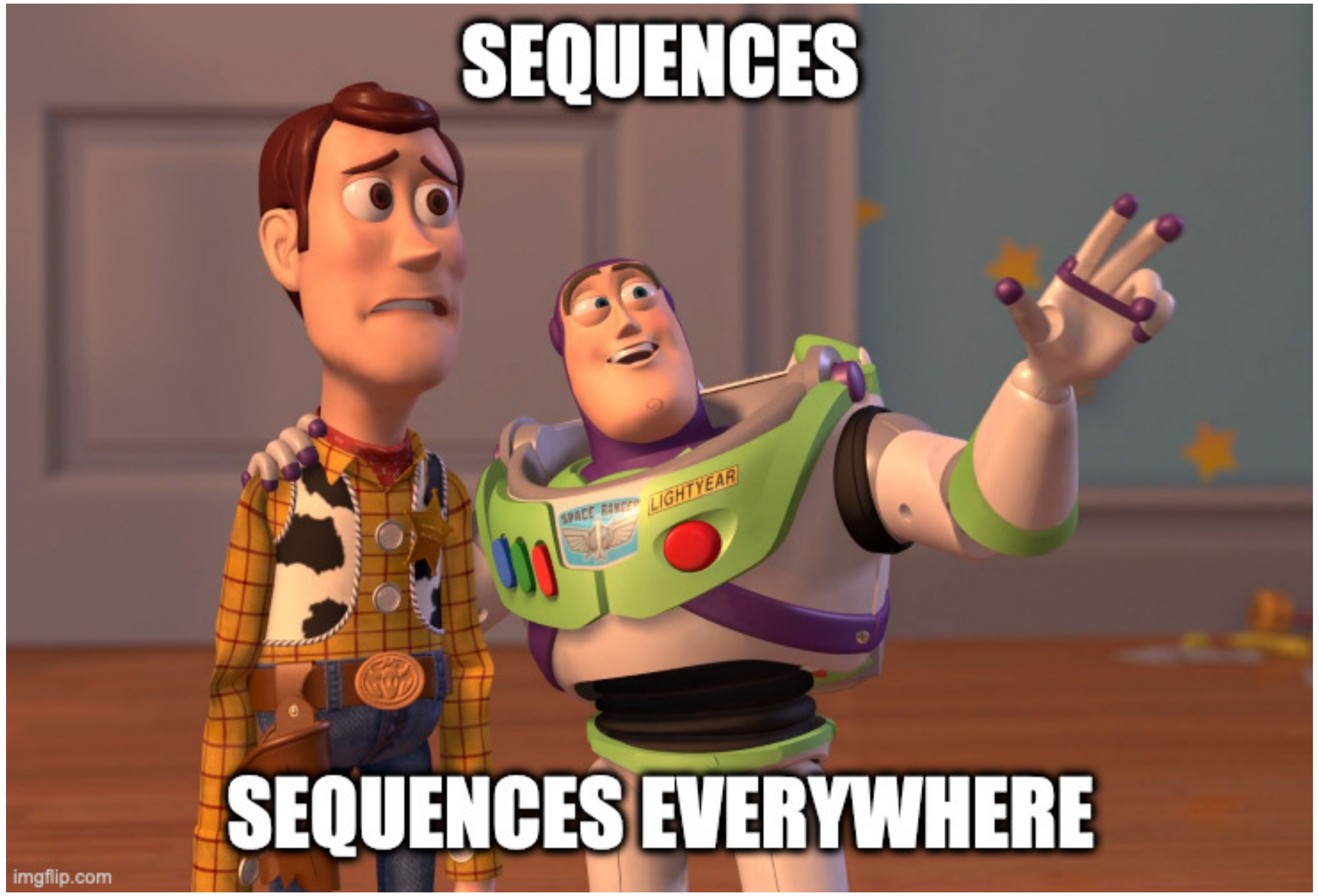
Sequences Everywhere

- `seq<'a>` is just an alias for `IEnumerable<'a>`
- Anything that implements `IEnumerable` is also a sequence
- strings, lists, arrays etc.



Sequences Everywhere

- `seq<'a>` is just an alias for `IEnumerable<'a>`
- Anything that implements `IEnumerable` is also a sequence
- strings, lists, arrays etc.




```
> Seq.zip [1;2;3] "abc";;

val it : seq<int * char> = seq [(1, 'a'); (2, 'b'); (3, 'c')]
```

Summary

- Sequences are a lot like lists, but more general (They have many of the same functions: fold, map, filter etc.)
- Sequences are more flexible
  - they can be infinite
  - elements can be computed on-the-fly (i.e. lazily)
- Sequence expressions allow for convenient manipulation and construction of sequences

Questions?