IT UNIVERSITY OF COPENHAGEN

Functional Programming

Lecture 2

Jesper Bengtson

Credit where credit is due

These slides are based on original slides by Michael R. Hansen at DTU.
Thank you!

The original slides have been used for a functional programming course at DTU

Last week

We introduced functional programming

- functions are first-class citizens
- Keep side effects to a minimum
- type inference vs type checking
- recursive functions rather than loops

This week

- Function composition
- Lists
- Polymorphism
- Higher-order functions
- More recursion
- Acquaint yourself with a major part of the F# language

Questions?

Some observations

Imperative languages are typically structured around statements and expressions

```
if (x > 4) then
    return “gt4”
else
    return “lt4”
```

is a statement

Some observations

Imperative languages are typically structured around statements and expressions

`return “gt4”` and `return “lt4”`

are statements

Some observations

Imperative languages are typically structured around statements and expressions

```
x > 4
```

is an expression

Some observations

Expressions and statements are not interchangeable

```
“result is: ” +
if (x > 4) then
    return “gt4”
else
    return “lt4”
```

is NOT a valid program

Some observations

Functional languages are more expression oriented

```
“result is: “ +
if (x > 4) then
   “gt4”
else
   “lt4”
```

IS a valid program

Some observations

As we will see, many things can be part of expressions

- Function declarations
- if-statements
- Matches
- Calculations
- …

Anonymous functions

Functions can be used as values or parameters to other functions just like standard integers or booleans

Anonymous functions

Functions can be used as values or parameters to other functions just like standard integers or booleans

There is NO difference
learn to appreciate this

Anonymous functions

Function expressions with general patterns

```
function
  | []          -> "Empty"
  | [x]         -> "One"
  | [x; y]      -> "Two"
  | [x; y; z]   -> "Three"
  | _           -> "Many"
```

returns a function that has type
α list -> string

Anonymous functions

Function expressions with general patterns

This function is polymorphic (α list) where α can be any type (int, bool, another list, …). F# writes ‘a, ‘b, ‘c in ascii where we write α, β, and ɣ in plain text

We will cover polymorphism shortly

returns a function that has type
α list -> string

Anonymous functions

Simple function expressions

```
fun r -> System.Math.PI * r * r
```

returns a function that has type
float -> float

Anonymous functions

Currying

```
fun x y z -> x + y + z
```

returns a function that has type
int -> int -> int -> int

Anonymous functions

```
fun x y z -> x + y + z
```

Anonymous functions

```
fun x y z -> x + y + z
```

is the same function as

```
fun x -> fun y -> fun z -> x + y + z
```

Anonymous functions

```
fun x y z -> x + y + z
```

is the same function as

```
fun x -> fun y -> fun z -> x + y + z
```

which is the same function as

```
fun x -> fun y z -> x + y + z
```

Anonymous functions

```
fun x y z -> x + y + z
```

is the same function as

```
fun x -> fun y -> fun z -> x + y + z
```

which is the same function as

```
fun x -> fun y z -> x + y + z
```

which is the same function as

```
fun x y -> fun z -> x + y + z
```

Function declarations

```
let f x = e
```

means

```
let f = fun x -> e
```

Function declarations

Conceptually NO different to other declarations

```
let a = 5
let b = true
let c = 'Q'
let d = (5, "meters")
let e = 2.71828
let f = fun x -> x + 3
```

Partial application

Suppose we have a cube with side length s (in meters), containing a liquid with density ρ (measured in kilograms per cube meter). The weight of the liquid is then given by $\rho * s^3$

```
let weight rho s = rho * (s ** 3.0)
```

This function has the type
float -> float -> float

Partial application

```
let weight rho s = rho * s ** 3.0

let methanolWeight = weight 786.5;;
val methanolWeight : float -> float

let waterWeight = weight 1000.0;;
val waterWeight : float -> float
```

Partial application

```
let weight rho s = rho * s ** 3.0

let methanolWeight = weight 786.5;;
val methanolWeight : float -> float
```

One and Two cubic meters of methanol
then weigh respectively

```
methanolWeight 1.0;;
val it : float = 786.5

methanolWeight 2.0;;
val it : float = 6292.0
```

kilograms

Patterns

We have seen how to use patterns when creating functions

```
function
| []          -> "Empty"
| [x]         -> "One"
| [x; y]      -> "Two"
| [x; y; z]   -> "Three"
| _           -> "Many"
```

Patterns

Patterns can also be used on their own, without creating a function

```
match lst with
| []         -> "Empty"
| [x]        -> "One"
| [x; y]     -> "Two"
| [x; y; z]  -> "Three"
| _          -> "Many"
```

Assuming the value lst is a list, this expression returns a string

Patterns

These two function expressions are identical

```
fun lst ->
    match lst with
     | []          -> “Empty" | [_] -> "One"
     | [_; _]      -> “Two”
     | [_; _; _] -> “Three" | _   -> "Many"

    function
     | []          -> “Empty" | [_] -> "One"
     | [_; _]      -> "Two"
     | [_; _; _] -> “Three" | _  -> "Many"
```

Infix operators

Infix operators ⊕ have prefix versions (⊕) and are curried

```
(+) : int -> int -> int
(-) : int -> int -> int
(*) : int -> int -> int
(/) : int -> int -> int
```

These all have overloaded versions for other (but always the same) numeric types

Infix operators

Infix operators ⊕ have prefix versions (⊕)
and are curried

```
(+) 5 6
```

is the same as

```
5 + 6
```

Infix operators

Infix operators can be partially applied

```
(+) 5
```

is the same as

```
fun x -> 5 + x
```

Infix operators

You can define your own infix operators

```
let (.+.) = fun x y -> x + y
```

or

```
let (.+.) x y = x + y
```

Infix operators

You can define your own infix operators

```
let (.+.) = fun x y -> x + y
```

You will get to define several of these for the next assignment

```
let (.+.) x y = x + y
```

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

```
g >> f
```

Function composition

For example

Assume f(x) = x + 3 and g(y) = y * y

$$(f \circ g)(z) = z * z + 3$$

In F# we can write

```
let h = fun y -> y*y >>
        fun x -> x+3
```

h 0 = 3 h 4 = 19 h 20 = 403

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

```
g >> f
```

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

g >> f or f << g

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

g >> f or f << g

The arrows point towards the outermost function

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

g >> f or f << g

i.e. the function that will run last

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

g >> f or f << g

```
let (>>) g f = fun x -> f (g x)
```

```
let (<<) f g = fun x -> f (g x)
```

Operators |> and <|

The operator |> means “send the value as argument to the function on the right”

`x |> f` is equivalent to `f x`

The operator <| means “send the value as argument to the function on the left”

`f <| x` is equivalent to `f x`

Operators |> and <|

The operator |> means “send the value as argument to the function on the right”

This just seems like wasted effort… `x |> f` is harder to read than `f x`

`f <| x` is equivalent to `f x`

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> fun x -> x*x |> (+) 3 = 19
```

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> fun x -> x*x |> (+) 3 = 19
```

Remember that

```
(+) 3
```

is the same as

```
fun x -> 3 + x
```

Operators |> and <|

While

```
x |> f
```

is not easier to read than

```
f x,
```

```
x |> f |> g |> h
```

is easier to read than

```
h (g (f x))
```

(especially when the functions are large or partially applied with many arguments) and naturally shows how x is transformed by the functions

Types and exceptions

It is possible to declare your own types and exceptions

As an example, recall that the formula for solving a second degree polynomial is:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Where $b^2 - 4ac$ is called the determinant

Types and exceptions

A quadratic formula has the form

$ax^2 + bx + c$

a solution has two possible values and both can be represented by tuples

```
type sdp = float * float * float
type solution = float * float
```

We will use exceptions to handle division by zero and negative square roots.

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
exception SolveSDP
```

Types and exceptions

```
let solve (a,b,c) =
    if b * b - 4.0 * a * c < 0.0 ||
       a = 0.0 then
      raise SolveSDP
    else
      ((-b + sqrt(b * b - 4.0 * a * c)) /
       (2.0 * a),

       (-b - sqrt(b * b - 4.0 * a * c)) /
       (2.0 * a));;
```

solve has type `sdp -> solution`

Types and exceptions

```
let solve (a,b,c) =
    i

    el
      ((-b + sqrt(b * b - 4.0 * a * c)) /
       (2.0 * a),

       (-b - sqrt(b * b - 4.0 * a * c)) /
       (2.0 * a));;
```

What is the problem with this solution?

solve has type `sdp -> solution`

Nested let-statements

Nested let-statements let us reuse results from previous computations

```
let solve(a,b,c) =
    let d = b * b - 4.0 * a * c
    if d < 0.0 || a = 0.0 then
      raise SolveSDP
    else
      ((-b + sqrt d) / (2.0 * a),
       (-b - sqrt d) / (2.0 * a));;
```

Nested let-statements

Nested let-statements let us reuse results from previous computations

```
let solve(a,b,c) =
  let sqrtD =
    let d = b * b - 4.0 * a * c
    if d < 0.0 || a = 0.0 then
      raise SolveSDP
    else
     sqrt d
  ((-b + sqrtD) / (2.0 * a),
   (-b - sqrtD) / (2.0 * a));;
```

Nested let-statements

Nested let-statements let us reuse results from previous computations

```
let so                                    c
  let
    le
    if                                    n

    el
      sqrt d
  ((-b + sqrtD) / (2.0 * a),
   (-b - sqrtD) / (2.0 * a));;
```

Indentation Matters!

Nested let-statements

Nested let-statements are also really useful for local functions

```
let power x n =
  let rec aux =
    function
    | 0 -> 1.0
    | m -> x * aux (m - 1)
  aux n
```

Records

Records allow us to structure data

```
type person =
    { firstname : string;
      lastname  : string;
      age       : int }
```

Records can be of arbitrary size

Records

Records are declared by providing input to all of their fields
(no partial application)

```
let Jesper =
    { firstname = "Jesper";
      lastname = "Bengtson";
      age = 41} ;;
val Jesper : person =
 { firstname = "Jesper";
    lastname = "Bengtson";
    age = 41;}
```

Records

Records are declared by providing input to all of their fields
(no partial application)

All field names must match with an already declared type

```
{ firstname = "Jesper";
  lastname = "Bengtson";
  age = 41;}
```

Records

Individual fields are accessed using their name

```
jesper.firstname;;
val it : string = "Jesper"

jesper.lastname;;
val it : string = "Bengtson"

jesper.age;;
val it : int = 41
```

let-patterns

It is possible to obtain values from compound statements

```
let { firstname = fn;
      lastname = ln;
      age = x} = jesper in
    (fn, ln, x);;
val it : string * string * int =
    (“Jesper", "Bengtson", 41)
```

let-patterns

It is possible to obtain values from compound statements

```
let (a, b) = (5, true) in (b, a);;
val it : bool * int = (true, 5)
```

Lists

We will expand on lists

- Recursion on lists
- Polymorphism
- Higher-order functions

List constructors

Recall hat lists are generated as follows

- [] is the empty list
- x::xs returns a list with head x and tail xs

```
> 5::3::-23::[];;
val it : int list = [5; 3; -23]
```

Length of a list

We recurse over the constructors

```
let rec length =
  function
  | [] -> 0
  | x::xs -> 1 + length xs
```

Length of a list

We recurse over the constructors

```
let rec length =
  function
  | [] -> 0
  | x::xs -> 1 + length xs
```

Has type α `list -> int`

Appending lists

We recurse over the constructors

```
let rec (@) xs ys =
  match xs with
  | []    -> ys
  | x::xs -> x :: (xs @ ys)
```

Has type `α list -> α list -> α list`

Higher-order functions

- Higher-order functions are functions that take other functions as arguments (we have seen a few, |>, <|, >> and <<, for instance)
- With lists these become very powerful and elegant
- What we show here is also available for other datatypes (maps, sets, arrays, …)

Maps

Patterns

Consider the following function

```
let rec posList =
    function
    | []    -> []
    | x::xs -> (x > 0) :: posList xs;;

posList : int list -> bool list

posList [4; -5; 6] =
        [true; false; true]
```

Patterns

and consider the following function

```
let rec addElems =
    function
    | [] -> []
    | (x, y) :: zs -> (x + y) :: addElems zs

addElems : (int * int) list -> int list

addElems [(1, 2); (3, 4); (5, 6)] = [3; 7; 11]
```

Patterns

They follow the same pattern - given a list

$$[x_1;\ x_2;\ \ldots;\ x_n]$$

and a function `f`, they return

$$[f\ x_1;\ f\ x_2;\ \ldots;\ f\ x_n]$$

Maps

`List.map` is a library function

Given a function f,

List.map f $[x_0; x_1; \ldots ; x_{n-1}]$

returns

$[f\ x_0; f\ x_1; \ldots ; f\ x_{n-1}]$

Maps with for loops

List.map f $[x_0; x_1; \ldots; x_{n-1}]$ returns $[f\ x_0; f\ x_1; \ldots; f\ x_{n-1}]$

In general a pseudo-code loop of this shape

```
ys = [];
for(int i = xs.Length - 1; i >= 0, i--) {
    ys = f (xs[i]) :: ys;
}
```

is replaced by map like this

```
let ys = List.map f xs
```

Maps

`List.map` is a library function

```
let rec map f =
    function
    | []     -> []
    | x :: xs -> f x :: map f xs
```

```
map : (‘a -> ‘b) -> ‘a list -> ‘b list
```

List.map f $[x_0; x_1; \ldots ; x_{n-1}] = [f\ x_0; f\ x_1; \ldots ; f\ x_{n-1}]$

Maps

```
let rec posList =
    function
    | []      -> []
    | x :: xs -> (x > 0) :: posList xs
```

can now be written as

```
let posList lst = map (fun x -> 0 < x) lst
```

Maps

```
let rec posList =
    function
    | []      -> []
    | x :: xs -> (x > 0) :: posList xs
```

can now be written as

```
let posList lst = map (fun x -> 0 < x) lst
```

or even shorter as: `let posList lst = map ((<) 0) lst`

Maps

```
let rec posList =
    function
    | []      -> []
    | x :: xs -> (x > 0) :: posList xs
```

can now be written as

```
let posList lst = map (fun x -> 0 < x) lst
```

or even shorter as: `let posList lst = map ((<) 0) lst`

or even shorter as: `let posList = map ((<) 0)`

Maps

```
let rec posList =
    function
    | []      -> []
    | x :: xs -> (x > 0) :: posList xs
```

can now be written as

```
let posList lst = map (fun x -> 0 < x) lst
```

or even shorter as: `let posList lst = map ((<) 0) lst`

or even shorter as: `let posList = map ((<) 0)`

Example: `posList [-5; 1; 0] = [false, true, false]`

Maps

```
let rec addElems =
    function
    | []              -> []
    | (x, y) :: zs -> (x + y) :: addElems zs
```

can now be written as

```
let addElems lst = map (fun (a, b) -> a + b) lst
```

Maps

```
let rec addElems =
    function
    | []              -> []
    | (x, y) :: zs -> (x + y) :: addElems zs
```

can now be written as

```
let addElems lst = map (fun (a, b) -> a + b) lst
```

or even shorter as

```
let addElems = map (fun (a, b) -> a + b)
```

Maps

```
let rec addElems =
    function
    | []              -> []
    | (x, y) :: zs -> (x + y) :: addElems zs
```

can now be written as

```
let addElems lst = map (fun (a, b) -> a + b) lst
```

or even shorter as

```
let addElems = map (fun (a, b) -> a + b)
```

Example:

```
addElems [(1, 2); (3, 4); (5, 6)] =
         [3; 7; 11]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ↝
map add [(1, 2); (3, 4); (5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)] ⇝
3 :: 3 + 4 :: map add [(5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)] ⇝
3 :: 3 + 4 :: map add [(5, 6)] ⇝ 3 :: 7 :: map add [(5, 6)]
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)] ⇝
3 :: 3 + 4 :: map add [(5, 6)] ⇝ 3 :: 7 :: map add [(5, 6)] ⇝
3 :: 7 :: add (5, 6) :: map add []
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)] ⇝
3 :: 3 + 4 :: map add [(5, 6)] ⇝ 3 :: 7 :: map add [(5, 6)] ⇝
3 :: 7 :: add (5, 6) :: map add [] ⇝
3 :: 7 :: 5 + 6 :: map add []
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)] ⇝
3 :: 3 + 4 :: map add [(5, 6)] ⇝ 3 :: 7 :: map add [(5, 6)] ⇝
3 :: 7 :: add (5, 6) :: map add [] ⇝
3 :: 7 :: 5 + 6 :: map add [] ⇝ 3 :: 7 :: 11 :: map add []
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)] ⇝
3 :: 3 + 4 :: map add [(5, 6)] ⇝ 3 :: 7 :: map add [(5, 6)] ⇝
3 :: 7 :: add (5, 6) :: map add [] ⇝
3 :: 7 :: 5 + 6 :: map add [] ⇝ 3 :: 7 :: 11 :: map add [] ⇝
3 :: 7 :: 11 :: []
```

Maps (evaluation)

```
let rec map f =
    function
    | []      -> []
    | x :: xs -> f x :: map f xs
```

```
let add (a, b) = a + b
let addElems lst = map add lst
```

```
addElems [(1, 2); (3, 4); (5, 6)] ⇝
map add [(1, 2); (3, 4); (5, 6)] ⇝
add (1, 2) :: map add [(3, 4); (5, 6)] ⇝
1 + 2 :: map add [(3, 4); (5, 6)] ⇝
3 :: map add [(3, 4); (5, 6)] ⇝
3 :: add (3, 4) :: map add [(5, 6)] ⇝
3 :: 3 + 4 :: map add [(5, 6)] ⇝ 3 :: 7 :: map add [(5, 6)] ⇝
3 :: 7 :: add (5, 6) :: map add [] ⇝
3 :: 7 :: 5 + 6 :: map add [] ⇝ 3 :: 7 :: 11 :: map add [] ⇝
3 :: 7 :: 11 :: [] ⇝ [3; 7; 11]
```

Filters

Patterns

Consider the following function

```
let rec removeNegative =
    function
    | []                        -> []
    | x :: xs when x >= 0 -> x :: removeNegative xs
    | _ :: xs                   -> removeNegative xs
```

```
removeNegative : int list -> int list
```

```
removeNegative [4; -5; 6; 0] = [4; 6]
```

Filters

For a predicate `p : 'a -> bool`,

filter p [x_1; x_2; …; x_n]

returns a list of all elements x_i such that `p` x_i `= true`

Filter with loops

filter p [x_1; x_2; …; x_n]

returns a list of all elements x_i such that `p` x_i `= true`

```
let ys = filter p xs
```

vs.

```
ys = [];
for(int i = xs.Length - 1; i >= 0; i--) {
    if (p (xs[i])) {
        ys = xs[i] :: ys;
    }
}
```

Filters

filter p [x_1; x_2; …; x_n]

returns a list of all elements x_i such that `p` x_i `= true`

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs               -> filter f xs
```

```
filter : (‘a -> bool) -> ‘a list -> ‘a list
```

Filters

filter p [x_1; x_2; …; x_n]

returns a list of all elements x_i such that `p` x_i `= true`

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs               -> filter f xs
```

```
filter : (‘a -> bool) -> ‘a list -> ‘a list
```

```
    let removeNegative = filter ((<=) 0)
```

```
      removeNegative [4; -5; 6; 0] = [4; 6; 0]
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs               -> filter f xs

let removeNegative = filter ((<=) 0)

removeNegative [4; -5; 6; 0]
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs               -> filter f xs

let removeNegative = filter ((<=) 0)
```

```
removeNegative [4; -5; 6; 0] ↝ filter ((<=) 0) [4; -5; 6; 0]
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs               -> filter f xs

let removeNegative = filter ((<=) 0)
```

```
removeNegative [4; -5; 6; 0] ↝ filter ((<=) 0) [4; -5; 6; 0] ↝
4 :: filter ((<=) 0) [-5; 6; 0]
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs               -> filter f xs

let removeNegative = filter ((<=) 0)
```

```
removeNegative [4; -5; 6; 0] ↝ filter ((<=) 0) [4; -5; 6; 0] ↝
4 :: filter ((<=) 0) [-5; 6; 0] ↝ 4 :: filter ((<=) [6; 0]
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs               -> filter f xs

let removeNegative = filter ((<=) 0)
```

```
removeNegative [4; -5; 6; 0] ↝ filter ((<=) 0) [4; -5; 6; 0] ↝
4 :: filter ((<=) 0) [-5; 6; 0] ↝ 4 :: filter ((<=) [6; 0] ↝
4 :: 6 :: filter ((<=) 0) [0]
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs              -> filter f xs

let removeNegative = filter ((<=) 0)
```

```
removeNegative [4; -5; 6; 0] ↝ filter ((<=) 0) [4; -5; 6; 0] ↝
4 :: filter ((<=) 0) [-5; 6; 0] ↝ 4 :: filter ((<=) [6; 0] ↝
4 :: 6 :: filter ((<=) 0) [0] ↝
4 :: 6 :: 0 :: filter ((<=) [])
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs              -> filter f xs

let removeNegative = filter ((<=) 0)
```

```
removeNegative [4; -5; 6; 0] ↝ filter ((<=) 0) [4; -5; 6; 0] ↝
4 :: filter ((<=) 0) [-5; 6; 0] ↝ 4 :: filter ((<=) [6; 0] ↝
4 :: 6 :: filter ((<=) 0) [0] ↝
4 :: 6 :: 0 :: filter ((<=) []) ↝ 4 :: 6 :: 0 :: []
```

Filters (evaluation)

```
let rec filter f =
    function
    | []                    -> []
    | x :: xs when f x -> x :: (filter f xs)
    | _ :: xs              -> filter f xs

let removeNegative = filter ((<=) 0)
```

```
removeNegative [4; -5; 6; 0] ↝ filter ((<=) 0) [4; -5; 6; 0] ↝
4 :: filter ((<=) 0) [-5; 6; 0] ↝ 4 :: filter ((<=) [6; 0] ↝
4 :: 6 :: filter ((<=) 0) [0] ↝
4 :: 6 :: 0 :: filter ((<=) []) ↝ 4 :: 6 :: 0 :: [] ↝
[4; 6; 0]
```

Exists

Patterns

Consider the following function

```
let rec containsPositive =
    function
    | []                    -> false
    | x :: _ when x > 0 -> true
    | _ :: xs               -> containsPositive xs
```

```
containsPositive : int list -> bool
```

```
containsPositive [-5; 4; 6; 0] = true
```

```
containsPositive [-5; -4; -6; 0] = false
```

Exists

For a predicate `p : 'a -> bool`,

exists p [x_1; x_2; …; x_n]

returns true if there is an element x_i such that p x_i holds, and false otherwise

Exists with loops

exists p [x_1; x_2; …; x_n]

returns true if there is an element x_i such that p x_i holds, and false otherwise

```
let b = exists p xs  vs.
```

```
b = false;
for(i = 0; i < xs.Length; i++) {
    b = b || p (xs[i]);
}
```

Exists

```
let rec exists p =
  function
  | []                  -> false
  | x :: _ when p x -> true
  | _ :: xs             -> exists p xs;;
exists : ('a -> bool) -> 'a list -> bool
```

Exists

```
let rec exists p =
  function
  | []                  -> false
  | x :: _ when p x -> true
  | _ :: xs             -> exists p xs;;
```

```
exists : ('a -> bool) -> 'a list -> bool
```

```
let containsPositive = exists ((<) 0)
```

```
containsPositive [-5; 4; 6; 0] = true
```

```
containsPositive [-5; -4; -6; 0] = false
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)

containsPositive [-5; 4; 6; 0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)

containsPositive [-5; 4; 6; 0] ⇝
exists ((<) 0) [-5; 4; 6; 0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)
```

```
containsPositive [-5; 4; 6; 0] ↝
exists ((<) 0) [-5; 4; 6; 0] ↝
exists [4; 6; 0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)
```

containsPositive [-5; 4; 6; 0] ⇝

exists ((<) 0) [-5; 4; 6; 0] ⇝

exists [4; 6; 0] ⇝ true

Exists (evaluation)

```
let rec exists p =
   function
   | []                -> false
   | x :: _ when p x -> true
   | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)

containsPositive [-5; -4; -6; 0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)

containsPositive [-5; -4; -6; 0] ⇝
exists ((<) 0) [-5; -4; -6; 0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)
```

```
containsPositive [-5; -4; -6; 0] ↝
exists ((<) 0) [-5; -4; -6; 0] ↝
exists ((<) 0) [-4; -6; 0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)
```

```
containsPositive [-5; -4; -6; 0] ↝
exists ((<) 0) [-5; -4; -6; 0] ↝
exists ((<) 0) [-4; -6; 0] ↝
exists ((<) 0) [-6; 0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)
```

```
containsPositive [-5; -4; -6; 0] ⇝
exists ((<) 0) [-5; -4; -6; 0] ⇝
exists ((<) 0) [-4; -6; 0] ⇝
exists ((<) 0) [-6; 0] ⇝
exists ((<) 0) [0]
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)
```

```
containsPositive [-5; -4; -6; 0] ⇝
exists ((<) 0) [-5; -4; -6; 0] ⇝
exists ((<) 0) [-4; -6; 0] ⇝
exists ((<) 0) [-6; 0] ⇝
exists ((<) 0) [0] ⇝
exists ((<) 0) []
```

Exists (evaluation)

```
let rec exists p =
  function
  | []                -> false
  | x :: _ when p x -> true
  | _ :: xs           -> exists p xs

let containsPositive = exists ((<) 0)
```

```
containsPositive [-5; -4; -6; 0] ⇝
exists ((<) 0) [-5; -4; -6; 0] ⇝
exists ((<) 0) [-4; -6; 0] ⇝
exists ((<) 0) [-6; 0] ⇝
exists ((<) 0) [0] ⇝
exists ((<) 0) [] ⇝ false
```

Forall

Patterns

Consider the following function

```
let rec allPositive =
    function
    | []                        -> true
    | x :: xs when x <= 0 -> false
    | x :: xs                   -> containsPositive xs
```

```
allPositive : int list -> bool
```

```
allPositive [5; -4; 6; 0] = false
```

```
allPositive [5; 4; 6; 1] = true
```

Forall

For a predicate `p : 'a -> bool`,

forall p [x_1; x_2; …; x_n]

returns true if p x_i holds for all elements x_i,
and false otherwise

Forall with loops

forall p [x_1; x_2; …; x_n]

returns true if p x_i holds for all elements x_i,
and false otherwise

```
let b = forall p xs
```

vs.

```
b = true;
for(i = 0; i < xs.Length; i++) {
    b = b && p (xs[i]);
}
```

Forall

```
let rec forall p =
  function
  | []                    -> true
  | x :: xs when p x -> forall p xs;;
  | _                     -> false
```

```
forall : ('a -> bool) -> 'a list -> bool
```

Forall

```
let rec forall p =
  function
  | []                    -> true
  | x :: xs when p x -> forall p xs;;
  | _                     -> false
```

```
forall : ('a -> bool) -> 'a list -> bool
```

```
let allPositive = forall ((<) 0)
```

```
allPositive [5; -4; 6; 0] = false
```

```
allPositive [5; 4; 6; 1] = true
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)

allPositive [5; -4; 6; 0]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             —> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; -4; 6; 0] ⇝
forall ((<) 0) [5; -4; 6; 0]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; -4; 6; 0] ↝
forall ((<) 0) [5; -4; 6; 0] ↝
forall [-4; 6; 0]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; -4; 6; 0] ⇝
forall ((<) 0) [5; -4; 6; 0] ⇝
forall [-4; 6; 0] ⇝ false
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)

allPositive [5; 4; 6; 1]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; 4; 6; 1] ⇝
forall ((<) 0) [5; 4; 6; 1]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; 4; 6; 1] ⇝
forall ((<) 0) [5; 4; 6; 1] ⇝
forall ((<) 0) [4; 6; 1]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; 4; 6; 1] ⇝
forall ((<) 0) [5; 4; 6; 1] ⇝
forall ((<) 0) [4; 6; 1] ⇝
forall ((<) 0) [6; 1]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; 4; 6; 1] ⇝
forall ((<) 0) [5; 4; 6; 1] ⇝
forall ((<) 0) [4; 6; 1] ⇝
forall ((<) 0) [6; 1] ⇝
forall ((<) 0) [1]
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; 4; 6; 1] ⇝
forall ((<) 0) [5; 4; 6; 1] ⇝
forall ((<) 0) [4; 6; 1] ⇝
forall ((<) 0) [6; 1] ⇝
forall ((<) 0) [1] ⇝
forall ((<) 0) []
```

Forall (evaluation)

```
let rec forall p =
  function
  | []                  -> true
  | x :: xs when p x -> forall p xs
  | x :: xs             -> false

let allPositive = forall ((<) 0)
```

```
allPositive [5; 4; 6; 1] ↝
forall ((<) 0) [5; 4; 6; 1] ↝
forall ((<) 0) [4; 6; 1] ↝
forall ((<) 0) [6; 1] ↝
forall ((<) 0) [1] ↝
forall ((<) 0) [] ↝ true
```

Folds

Before we start, folds are what comes the closest to the standard loops that you are used to, even though this is not always directly apparent. They are exceptionally useful.

Folds (loops)

For a function `f`, and an initial value `acc`,

$$\texttt{fold f acc [}x_1\texttt{; }x_2\texttt{; …; }x_n\texttt{]}$$

returns

$$\texttt{f (… (f (f acc }x_1\texttt{) }x_2\texttt{) … }x_{n-1}\texttt{) }x_n$$

This particular fold loops over the elements of the list in order

Folds (loops)

fold f acc [x_1; x_2; …; x_n]

returns

f (… (f (f acc x_1) x_2) … x_{n-1}) x_n

```
let acc = fold f init xs
```

vs.

```
acc = init;
for(i = 0; i < xs.Length; i++)
{
    acc = f (acc, xs[i]);
}
```

Folds (loops)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold : (‘b -> ‘a –> ‘b) -> ‘b -> ‘a list -> ‘b
```

```
fold (-) 0 [1; 2; 3] = ((0-1)-2)-3 = -6
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs

fold (-) 0 [1; 2; 3]
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3]
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3]
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3]
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3] ⇝
fold (-) ((-) -1 2) [3]
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3] ⇝
fold (-) ((-) -1 2) [3] ⇝
fold (-) (-1 - 2) [3]
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3] ⇝
fold (-) ((-) -1 2) [3] ⇝
fold (-) (-1 - 2) [3] ⇝ fold (-) -3 [3]
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3] ⇝
fold (-) ((-) -1 2) [3] ⇝
fold (-) (-1 - 2) [3] ⇝ fold (-) -3 [3] ⇝
fold (-) ((-) -3 3) []
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3] ⇝
fold (-) ((-) -1 2) [3] ⇝
fold (-) (-1 - 2) [3] ⇝ fold (-) -3 [3] ⇝
fold (-) ((-) -3 3) [] ⇝
fold (-) (-3 - 3) []
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3] ⇝
fold (-) ((-) -1 2) [3] ⇝
fold (-) (-1 - 2) [3] ⇝ fold (-) -3 [3] ⇝
fold (-) ((-) -3 3) [] ⇝
fold (-) (-3 - 3) [] ⇝ fold (-) -6 []
```

Folds (evaluation)

```
let rec fold f acc =
    function
    | []      -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold (-) 0 [1; 2; 3] ⇝
fold (-) ((-) 0 1) [2; 3] ⇝
fold (-) (0 - 1) [2; 3] ⇝ fold (-) -1 [2; 3] ⇝
fold (-) ((-) -1 2) [3] ⇝
fold (-) (-1 - 2) [3] ⇝ fold (-) -3 [3] ⇝
fold (-) ((-) -3 3) [] ⇝
fold (-) (-3 - 3) [] ⇝ fold (-) -6 [] ⇝
-6
```

Folds (loops)

For a function `f`, and an initial value `acc`,

$$\texttt{foldBack f [}x_1\texttt{; }x_2\texttt{; …; }x_n\texttt{] acc}$$

returns

$$\texttt{f }x_1\texttt{ (f }x_2\texttt{ (… f }x_{n-1}\texttt{ (f }x_n\texttt{ acc)…))}$$

This particular fold is called a foldBack in F# because it starts at the end of the list

Folds (loops)

foldBack f [x_1; x_2; …; x_n] acc

returns

f x_1 (f x_2 (… f x_{n-1} (f x_n acc)…))

```
let acc = foldBack f xs init
```

vs.

```
acc = init;
for(i = xs.Length - 1; i >= 0; i--)
{
    acc = f (xs[i], acc);
}
```

Folds (loops)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack : (‘a -> ‘b –> ‘b) -> ‘a list -> ‘b -> ‘b
```

Folds (loops)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack : (‘a -> ‘b –> ‘b) -> ‘a list -> ‘b -> ‘b
```

```
foldBack (-) [1; 2; 3] 0 = 1-(2-(3-0)) = 2
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)

foldBack (-) [1; 2; 3] 0
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0)
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0))
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0)))
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0))) ⇝
(-) 1 ((-) 2 ((-) 3 0))
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0))) ⇝
(-) 1 ((-) 2 ((-) 3 0)) ⇝ (-) 1 ((-) 2 (3 - 0))
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0))) ⇝
(-) 1 ((-) 2 ((-) 3 0)) ⇝ (-) 1 ((-) 2 (3 - 0)) ⇝
(-) 1 ((-) 2 3)
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0))) ⇝
(-) 1 ((-) 2 ((-) 3 0)) ⇝ (-) 1 ((-) 2 (3 - 0)) ⇝
(-) 1 ((-) 2 3) ⇝ (-) 1 (2 - 3)
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0))) ⇝
(-) 1 ((-) 2 ((-) 3 0)) ⇝ (-) 1 ((-) 2 (3 - 0)) ⇝
(-) 1 ((-) 2 3) ⇝ (-) 1 (2 - 3) ⇝ (-) 1 -1
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0))) ⇝
(-) 1 ((-) 2 ((-) 3 0)) ⇝ (-) 1 ((-) 2 (3 - 0)) ⇝
(-) 1 ((-) 2 3) ⇝ (-) 1 (2 - 3) ⇝ (-) 1 -1 ⇝ 1 - -1
```

FoldBack (evaluation)

```
let rec foldBack f xs acc =
    match xs with
    | []      -> acc
    | x :: xs -> f x (foldBack f xs acc)
```

```
foldBack (-) [1; 2; 3] 0 ⇝
(-) 1 (foldBack (-) [2; 3] 0) ⇝
(-) 1 ((-) 2 (foldBack (-) [3] 0)) ⇝
(-) 1 ((-) 2 ((-) 3 (foldBack (-) [] 0))) ⇝
(-) 1 ((-) 2 ((-) 3 0)) ⇝ (-) 1 ((-) 2 (3 - 0)) ⇝
(-) 1 ((-) 2 3) ⇝ (-) 1 (2 - 3) ⇝ (-) 1 -1 ⇝ 1 - -1 ⇝ 2
```

Quote from student towards the end of the course (paraphrased)

"I just realised that with folds you can do absolutely anything!"

Quote from student towards the end of the course (paraphrased)

"I just realised that with folds you can do absolutely anything!"

Not quite true, but the TAs were dancing

With folds you can do almost anything

```
map f lst ==
    foldBack (fun x acc -> f x :: acc) lst []
filter p lst ==
    foldBack (fun x acc -> if p x then x :: acc else acc)
    lst []

exists p lst == fold (fun acc x -> p x || acc) false lst
forall p lst == fold (fun acc x -> p x && acc) true ls
```

With folds you can do almost anything

```
map f lst ==
    foldBack (fun x acc -> f x :: acc) lst []
filter p lst ==
    foldBack (fun x acc -> if p x then x :: acc else acc)
    lst []

exists p lst == fold (fun acc x -> p x || acc) false lst
forall p lst == fold (fun acc x -> p x && acc) true ls

exists p lst == foldBack (fun x acc -> p x || acc) lst false
forall p lst == foldBack (fun x acc -> p x && acc) lst true
```

With folds you can do almost anything

```
map f lst ==
    foldBack (fun x acc -> f x :: acc) lst []
filter p lst ==
    foldBack (fun x acc -> if p x then x :: acc else acc)
    lst []

exists p lst == fold (fun acc x -> p x || acc) false lst
forall p lst == fold (fun acc x -> p x && acc) true ls

exists p lst == foldBack (fun x acc -> p x || acc) lst false
forall p lst == foldBack (fun x acc -> p x && acc) lst true

exists p == foldBack (p >> (||)) false
forall p == foldBack (p >> (&&)) true
```

With folds you can do almost anything

This last one may seem magical

```
forall p lst == foldBack (fun x acc -> p x && acc) lst true

forall p == foldBack (p >> (&&)) true
```

With folds you can do almost anything

This last one may seem magical

```
p >> (&&)
```

```
forall p lst == foldBack (fun x acc -> p x && acc) lst true
```

```
forall p == foldBack (p >> (&&)) true
```

With folds you can do almost anything

This last one may seem magical

```
f >> g = fun x -> g (f x)

p >> (&&) ==
fun x => (&&) (p x)
```

```
forall p lst == foldBack (fun x acc -> p x && acc) lst true

forall p == foldBack (p >> (&&)) true
```

With folds you can do almost anything

This last one may seem magical

```
p >> (&&) ==
fun x => (&&) (p x) ==
fun x acc => (&&) (p x) acc
```

```
forall p lst == foldBack (fun x acc -> p x && acc) lst true
```

```
forall p == foldBack (p >> (&&)) true
```

With folds you can do almost anything

This last one may seem magical

```
p >> (&&) ==
fun x => (&&) (p x) ==
fun x acc => (&&) (p x) acc ==
fun x acc => (p x) && acc
```

```
forall p lst == foldBack (fun x acc -> p x && acc) lst true
```

```
forall p == foldBack (p >> (&&)) true
```

Piping and HoFs

We can do great things with piping, function composition and higher-order functions

Recall our equation solver

```
let solve (a, b, c) =
  let sqrtD =
    let d = b * b - 4.0 * a * c
    if d < 0.0 || a = 0.0 then
      raise SolveSDP
    else
     sqrt d
  ((-b + sqrtD) / (2.0 * a),
   (-b - sqrtD) / (2.0 * a));;
```

Piping and HoFs

Create a function that given a list of second degree polynomials (as defined before), return the smallest non-negative root smaller than 100 (assuming one exists).

Piping and HoFs

Create a function that given a list of second degre), return th aller than 100 (assuming one exists).

Start with the list of polynomials

```
sdps
```

Current result type: `(float * float * float) list`

Piping and HoFs

Create a function that given a list of second degre), return th aller than 100 (assuming one exists).

Solve the polynomials

```
sdps |>
List.map solve
```

Current result type: `(float * float) list`

Piping and HoFs

Create a function that given a list of second degr), return aller than 100 (assuming one exists).

Flatten the list of solutions - rather than a list of pairs of roots return a list of roots

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) []
```

Current result type: `float list`

Piping and HoFs

Create a function that given a list of second degr), return aller than 100 (assuming one exists).

Remove all roots smaller than zero

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0)
```

Current result type: `float list`

Piping and HoFs

Create a function that given a list of second degr), return aller than 100 (assuming one exists).

Return the smallest root that is smaller than 100.0

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

Current result type: float

Piping and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

is much easier to read than

```
List.fold min 100.0
  (List.filter ((<) 0.0)
    (List.fold (fun acc (a, b) -> a::b::acc) []
               (List.map solve sdps)))
```

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

… and is identical to

```
(List.map solve >>
 List.fold (fun acc (a, b) -> a::b::acc) [] >>
 List.filter ((<) 0.0) >>
 List.fold min 100.0) sdps
```

Questions?