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

Questions?

This week

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

Everything we do from now on depends on this lecture. Come back to it often. Study it often. Do not wait until the exam to catch up on things you might have missed.

Statements and expressions

Imperative languages are typically structured around statements and expressions

Statements

- do things (assign values to variables, print to screen, update memory, …)
- are built using statements and expressions

Expressions

- are evaluated to values
- are built using expressions only

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

This entire block of code is a statement consisting of an assignment and a conditional

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

The assignment is a statement that assigns the value 4 to the variable y

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

This if-expresion is a an expression that evaluates to the string `"gt4"` if x > y and `"lt4"` otherwise

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

The return statement exits the current method and sets the return value to "gt4"

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

The return statement exits the current method and sets the return value to `"lt4"`

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

4 is an expression that evaluates to the integer 4

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

`x > y` is an expression that evaluates to `true` if `x` is strictly greater than `y` and `false` otherwise

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

"gt4" is an expression that evaluates to the string "gt4"

Statements and expressions

Imperative languages are typically structured around statements and expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

"lt4" is an expression that evaluates to the string "lt4"

Statements and expressions

Functional languages **only** use expressions

Java

```
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

F#

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Functional languages **only** use expressions

This entire block of code is a an expression that evaluates to the string `"gt4"` if x > 4 and `"lt4"` otherwise

F#

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Functional languages **only** use expressions

This if-statement is a an expression that returns the string `"gt4"` if x > 4 and `"lt4"` otherwise

F#

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Functional languages **only** use expressions

4 is an expression that evaluates to the integer 4

F#

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Functional languages **only** use expressions

`x > y` is an expression that evaluates to `true` if `x` is strictly greater than `y` and `false` otherwise

No parentheses needed!

F#

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Functional languages **only** use expressions

"gt4" is an expression that evaluates to the string "gt4"

F#

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Functional languages **only** use expressions

`"lt4"` is an expression that evaluates to the string `"lt4"`

F#

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Expressions evaluate to a single value

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

Statements and expressions

Expressions evaluate to a single value

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

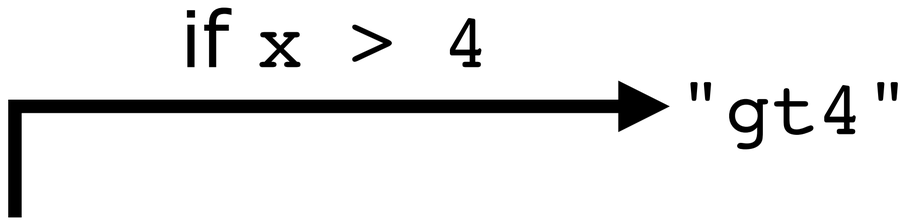
```
if x > 4 then
    "gt4"
else
    "lt4"
```

Statements and expressions

Expressions evaluate to a single value

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

```
if x > 4 then
    "gt4"
else
    "lt4"
```

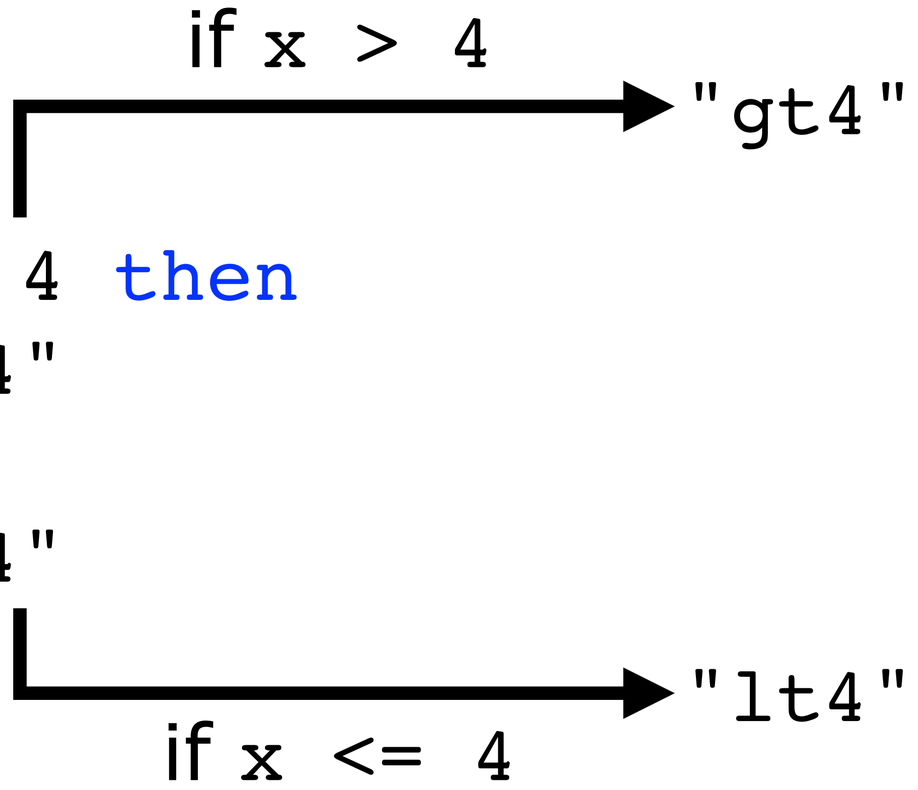


Statements and expressions

Expressions evaluate to a single value

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

```
if x > 4 then
    "gt4"
else
    "lt4"
```



Statements and expressions

What is the type of this expression?

```
let y = 4
if x > y then
    "gt4"
else
    "lt4"
```

- int
- string
- int -> string
- int -> int -> string

Statements and expressions

Is this possible in either language?

Java

```
"result is: " +
y = 4;
if (x > y) {
    return "gt4";
}
else {
    return "lt4";
}
```

F#

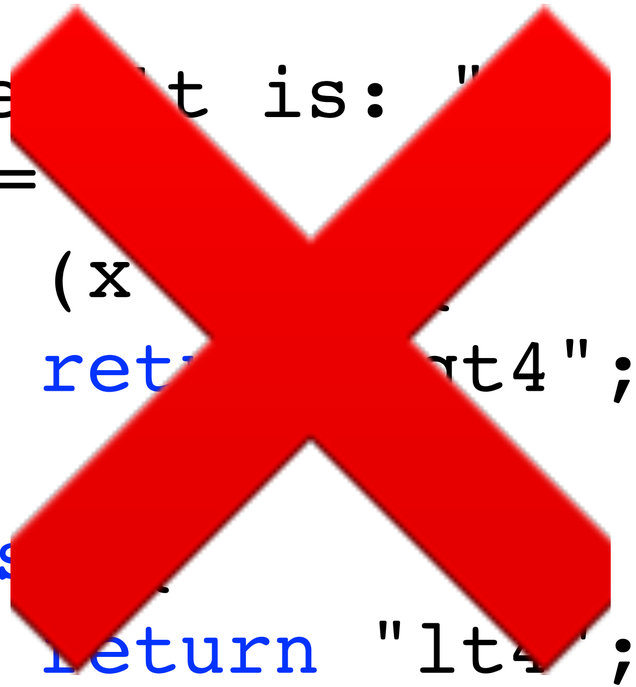
```
"result is: " +
let y = 4
if (x > y) then
    "gt4"
else
    "lt4"
```

Statements and expressions

Is this possible in either language?

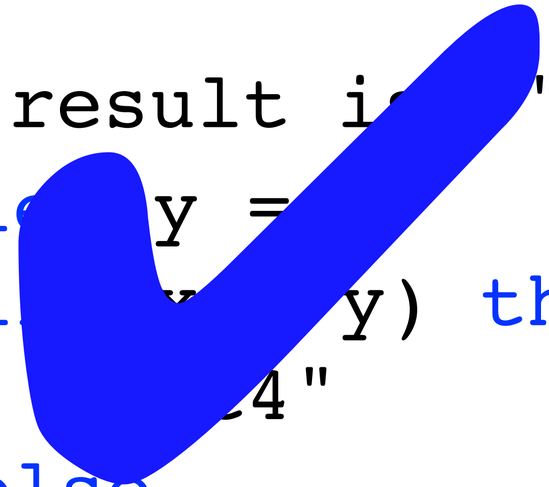
Java

```
"re    t is: "
y =
if (x
    ret      t4";
}
els
    return "lt4";
}
```



F#

```
"result i     " +
l    y =
i      y) then
      4"
else
    "lt4"
```



Statements and expressions

if-expressions in functional languages are more like the the ternary operator (<boolean> ? <val1> : <val2>) in Java

Java

```
y = 4;
"result is: " +
(x > y ? "gt4" : "lt4");
```

F#

```
"result is: " +
let y = 4
if (x > y) then
    "gt4"
else
    "lt4"
```

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
```

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
```

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

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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))
```

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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))
```

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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))
```

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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))
```

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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))
```

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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))
```

What's the problem with this solution?

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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 let us reuse results from previous computations

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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 let us reuse results from previous computations

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
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))
```

let-statements can have multi-level nesting

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

`solve` has type `sdp -> solution`

Types and exceptions

```
type sdp      = float * float * float
type solution = float * float
exception SolveSDP
let solv
  let sq
    let
    if d                                          SDP
    else
  ((-b +
   (-b -  sqrtD) / (2.0 * a))
```

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

`solve` has type `sdp -> solution`

Indentation matters!!!

Nested let-statements

Nested let-statements are useful for local functions

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

Nested let-statements

Nested let-statements are useful for local functions

```
power 2.0 3
```

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

Nested let-statements

Nested let-statements are useful for local functions

```
                                power 2.0 3 ↪
                                aux 3
let power x n =
  let rec aux =
    function
    | 0 -> 1.0
    | m ->
      x * aux (m - 1)
  aux n
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ↪
aux 3 ↪
2.0 * aux (3 - 1)
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ↪
aux 3 ↪
2.0 * aux (3 - 1) ↪
2.0 * aux 2
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ⇝
aux 3 ⇝
2.0 * aux (3 - 1) ⇝
2.0 * aux 2 ⇝
2.0 * (2.0 * aux (2 - 1))
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ↪
aux 3 ↪
2.0 * aux (3 - 1) ↪
2.0 * aux 2 ↪
2.0 * (2.0 * aux (2 - 1)) ↪
2.0 * (2.0 * aux 1)
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ↪
aux 3 ↪
2.0 * aux (3 - 1) ↪
2.0 * aux 2 ↪
2.0 * (2.0 * aux (2 - 1)) ↪
2.0 * (2.0 * aux 1) ↪
2.0 * (2.0 * (2.0 * aux (1 - 1)))
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ↪
aux 3 ↪
2.0 * aux (3 - 1) ↪
2.0 * aux 2 ↪
2.0 * (2.0 * aux (2 - 1)) ↪
2.0 * (2.0 * aux 1) ↪
2.0 * (2.0 * (2.0 * aux (1 - 1))) ↪
2.0 * (2.0 * (2.0 * aux 0)))
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ↪
aux 3 ↪
2.0 * aux (3 - 1) ↪
2.0 * aux 2 ↪
2.0 * (2.0 * aux (2 - 1)) ↪
2.0 * (2.0 * aux 1) ↪
2.0 * (2.0 * (2.0 * aux (1 - 1))) ↪
2.0 * (2.0 * (2.0 * aux 0))) ↪
2.0 * (2.0 * (2.0 * 1.0)))
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ↪
aux 3 ↪
2.0 * aux (3 - 1) ↪
2.0 * aux 2 ↪
2.0 * (2.0 * aux (2 - 1)) ↪
2.0 * (2.0 * aux 1) ↪
2.0 * (2.0 * (2.0 * aux (1 - 1))) ↪
2.0 * (2.0 * (2.0 * aux 0))) ↪
2.0 * (2.0 * (2.0 * 1.0))) ↪
2.0 * (2.0 * 2.0))
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ⇝
aux 3 ⇝
2.0 * aux (3 - 1) ⇝
2.0 * aux 2 ⇝
2.0 * (2.0 * aux (2 - 1)) ⇝
2.0 * (2.0 * aux 1) ⇝
2.0 * (2.0 * (2.0 * aux (1 - 1))) ⇝
2.0 * (2.0 * (2.0 * aux 0)) ⇝
2.0 * (2.0 * (2.0 * 1.0)) ⇝
2.0 * (2.0 * 2.0) ⇝
2.0 * 4.0
```

Nested let-statements

Nested let-statements are useful for local functions

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

```
power 2.0 3 ⤳
aux 3 ⤳
2.0 * aux (3 - 1) ⤳
2.0 * aux 2 ⤳
2.0 * (2.0 * aux (2 - 1)) ⤳
2.0 * (2.0 * aux 1) ⤳
2.0 * (2.0 * (2.0 * aux (1 - 1))) ⤳
2.0 * (2.0 * (2.0 * aux 0))) ⤳
2.0 * (2.0 * (2.0 * 1.0))) ⤳
2.0 * (2.0 * 2.0)) ⤳
2.0 * 4.0 ⤳
8.0
```

Lists

We will expand on lists

- Recursion on lists
- Polymorphism
- Higher-order functions

List constructors

Recall that 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
- With lists these become very powerful and elegant
- What we show here is also available for other datatypes (maps, sets, arrays, …)

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int
```

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int
```

The type of `square` is relatively straightforward. It takes an integer and returns an integer

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int
```

What about compute?

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int -> int
```

- `int -> int -> int`
- `int -> (int -> int)`
- `(int -> int) -> int`

Are these types different?

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int
```

- `int -> int -> int`
- `int -> (int -> int)`
- `(int -> int) -> int`

These are equal - the function takes two integers and returns an integer

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int
```

- `int -> int -> int`
- `int -> (int -> int)`
- `(int -> int) -> int`

This function takes a function, from int to int, and returns an int

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int

compute square 5 6
```

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int

compute square 5 6 ⇝
square 5 + 6
```

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int

compute square 5 6 ⇝
square 5 + 6 ⇝
(5 * 5) + 6
```

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int

compute square 5 6 ⇝
square 5 + 6 ⇝
(5 * 5) + 6 ⇝
25 + 6
```

Higher-order functions

```
let square x = x * x
let compute operator a b = operator a + b

square  : int -> int
compute : (int -> int) -> int -> int

compute square 5 6 ⇝
square 5 + 6 ⇝
(5 * 5) + 6 ⇝
25 + 6 ⇝
31
```

Maps

Patterns

Consider you want to transform one list into another list

- Square each element
- Add 42 to each element
- Calculate the square root of each element
- …

Squaring a list in Java and F#

```java
public static List<Integer> squareList (List<Integer> input) {
    List<Integer> result = new ArrayList<>();

    for(int el : input) { result.add(el * el); }

    return result;
}
```

Squaring a list in Java and F#

```java
public static List<Integer> squareList (List<Integer> input) {
    List<Integer> result = new ArrayList<>();

    for(int el : input) { result.add(el * el); }

    return result;
}
```

```
let rec squareList =
  function
  | [] -> []
  | x::xs -> x * x :: (squareList xs)
```

Squaring a list in Java and F#

```java
public static List<Integer> squareList (List<Integer> input) {
    List<Integer> result = new ArrayList<>();

    for(int el : input) { result.add(el * el); }

    return result;
}
```

```
let rec squareList =
  function
  | [] -> []
  | x::xs -> x * x :: (squareList xs)
```

squareList has type int list -> int list

Adding 42 to a list in Java and F#

```java
public static List<Integer> add42List (List<Integer> input) {
    List<Integer> result = new ArrayList<>();

    for(int el : input) { result.add(el + 42); }

    return result;
}
```

```
let rec add42List =
  function
  | [] -> []
  | x::xs -> x + 42 :: (add42List xs)
```

add42List has type int list -> int list

Merging the functions

```
public static <T, U> List<U> map (Function<T, U> mapper,
                                  List<T> input) {
    List<U> result = new ArrayList<>();

    for(T el : input) { result.add(mapper.apply(el)); }

    return result;
}
```

Merging the functions

```java
public static <T, U> List<U> map (Function<T, U> mapper,
                                  List<T> input) {
    List<U> result = new ArrayList<>();

    for(T el : input) { result.add(mapper.apply(el)); }

    return result;
}
```

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

`map` has type `('a -> 'b) -> 'a list -> 'b list`

Using maps in Java

```
public static <T, U> List<U> map (Function<T, U> mapper,
                                  List<T> input) {
    List<U> result = new ArrayList<>();
    for(T el : input) { result.add(mapper.apply(el)); }
    return result;
}

public static int square (int x) { return x * x; }
public static int add42  (int x) { return x + 42; }

public static List<Integer> squareList(List<Integer> input) {
    return map(Lecture2::square, input);
}

public static List<Integer> add42List(List<Integer> input) {
    return map(Lecture2::add42, input);
}
```

Using maps in F#

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

let square x = x * x
let add42  x = x + 42

let squareList lst = map square lst
let add42List  lst = map add42 lst
```

Using maps in F#

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

let square x = x * x
let add42  x = x + 42

let squareList lst = map square lst
let add42List  lst = map add42 lst
```

squareList [1; 2; 3; 4; 5] = [1; 4; 9; 16; 25]

add42List [1; 2; 3; 4; 5] = [43; 44; 45; 46; 47]

Streams in Java

Java has supports for streams which is the preferred way of working with higher-order functions in Java

```
public static int square (int x) { return x * x; }
public static int add42. (int x) { return x + 42; }

public static List<Integer> squareList(List<Integer> input) {
    return input.stream().map(Lecture2::square).toList();
}

public static List<Integer> add42List(List<Integer> input) {
    return input.stream().map(Lecture2::add42).toList();
}
```

Maps in the list library

`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 (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 you want to remove elements from a list

- Remove all even numbers
- Remove all odd numbers
- Remove all vehicles with two wheels
- ...

Remove odd numbers in Java and F#

```
    public static List<Integer> keepEven (List<Integer> input) {
        List<Integer> result = new ArrayList<>();

        for(int el : input) { if (el % 2 == 0) { result.add(el); } }

        return result;
    }
```

Remove odd numbers in Java and F#

```java
public static List<Integer> keepEven (List<Integer> input) {
    List<Integer> result = new ArrayList<>();

    for(int el : input) { if (el % 2 == 0) { result.add(el); } }

    return result;
}
```

```fsharp
let rec keepEven =
  function
  | []                          -> []
  | x::xs when x % 2 = 0 -> x :: (keepEven xs)
  | x::xs                       -> keepEven xs
```

Remove odd numbers in Java and F#

```java
public static List<Integer> keepEven (List<Integer> input) {
    List<Integer> result = new ArrayList<>();

    for(int el : input) { if (el % 2 == 0) { result.add(el); } }

    return result;
}
```

```
let rec keepEven =
  function
  | []                          -> []
  | x::xs when x % 2 = 0 -> x :: (keepEven xs)
  | x::xs                       -> keepEven xs
```

keepEven has type `int list -> int list`

Remove even numbers in Java and F#

```java
public static List<Integer> keepOdd (List<Integer> input) {
    List<Integer> result = new ArrayList<>();

    for(int el : input) { if (el % 2 == 1) { result.add(el); } }

    return result;
}
```

```
let rec keepOdd =
  function
  | []                          -> []
  | x::xs when x % 2 = 1 -> x :: (keepOdd xs)
  | x::xs                       -> keepOdd xs
```

keepOdd has type `int list -> int list`

The filter function

```
let rec keepOdd =
  function
  | []                          -> []
  | x::xs when x % 2 = 1 -> x :: (keepOdd xs)
  | x::xs                       -> keepOdd xs
```

This function can be generalised

```
let rec filter predicate =
  function
  | []                          -> []
  | x::xs when predictate x -> x :: (keepOdd xs)
  | x::xs                       -> keepOdd xs
```

filter has type `('a -> bool) -> 'a list -> 'a list`

The filter function

```
let rec filter predicate =
  function
  | []                              -> []
  | x::xs when predictate x -> x :: (keepOdd xs)
  | x::xs                           -> keepOdd xs
```

filter has type `('a -> bool) -> 'a list -> 'a list`

```
keepEven lst = filter (fun x -> x % 2 = 0) lst
keepOdd  lst = filter (fun x -> x % 2 = 1) lst
```

The filter function

```
let rec filter predicate =
  function
  | []                              -> []
  | x::xs when predictate x -> x :: (keepOdd xs)
  | x::xs                           -> keepOdd xs
```

filter has type `('a -> bool) -> 'a list -> 'a list`

```
keepEven lst = filter (fun x -> x % 2 = 0) lst
keepOdd  lst = filter (fun x -> x % 2 = 1) lst

keepEven [1; 2; 3; 4; 5] = [2; 4]
keepOdd  [1; 2; 3; 4; 5] = [1; 3; 5]
```

Filters in the list library

`List.filter` is a library function

```
let rec filter predicate =
  function
  | []                          -> []
  | x::xs when predictate x -> x :: (keepOdd xs)
  | x::xs                       -> keepOdd xs
```

`filter` has type `('a -> bool) -> 'a list -> 'a list`

filter pred [x_1; x_2; …; x_n]

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

Filters (evaluation)

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

let removeNegative = filter ((<=) 0)

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

Filters (evaluation)

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

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

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

Filters (evaluation)

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

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)

The following Java program sums all elements of a list

```
    public static int sumList(List<Integer> input) {
        int accumulator = 0;
        for(int el : input) { accumulator += el; }
        return accumulator;
    }
```

Folds (loops)

The following Java program sums all elements of a list

```java
public static int sumList(List<Integer> input) {
    int accumulator = 0;
    for(int el : input) { accumulator += el; }
    return accumulator;
}
```

and the following does the same in F#

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

Folds (loops)

The following Java program sums all elements of a list

```java
public static int sumList(List<Integer> input) {
    int accumulator = 0;
    for(int el : input) { accumulator += el; }
    return accumulator;
}
```

Another version that follows the loop a bit more closely

```
let sumList lst =
    let rec loop accumulator =
        function
        | []      -> accumulator
        | el::xs -> loop (accumulator + el) xs
    loop 0 lst
```

Folds (loops)

```
let sumList lst =
    let rec loop accumulator =
        function
        | []      -> accumulator
        | el::xs -> loop (accumulator + el) xs
    loop 0 lst
```

More generally

```
let rec fold folder accumulator =
    function
    | []      -> accumulator
    | el::xs -> fold folder (folder accumulator el) xs
```

Folds (loops)

```
let sumList lst =
    let rec loop accumulator =
        function
        | []      -> accumulator
        | el::xs -> loop (accumulator + el) xs
    loop 0 lst
```

More generally

```
let rec fold folder accumulator =
    function
    | []      -> accumulator
    | el::xs -> fold folder (folder accumulator el) xs

sumList lst = fold (fun a b -> a + b) 0 lst
```

Folds (loops)

```
let rec fold folder accumulator =
    function
    | []     -> accumulator
    | el::xs -> fold folder (folder accumulator el) xs

sumList lst = fold (fun a b -> a + b) 0 lst
```

sumList [1;2;3;4;5] = 0 + 1 + 2 + 3 + 4 + 5 = 15

Folds (loops)

```
let rec fold folder accumulator =
    function
    | []     -> accumulator
    | el::xs -> fold folder (folder accumulator el) xs
```

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

Folds (loops)

```
let rec fold folder accumulator =
    function
    | []     -> accumulator
    | el::xs -> fold folder (folder accumulator el) xs
```

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

Mathematically

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

returns

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

Folds (evaluation)

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

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

Folds (evaluation)

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

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

Folds (evaluation)

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

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

Folds (evaluation)

```
let rec fold folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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 folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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 folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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 folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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 folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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 folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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 folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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 folder accumulator =
    function
    | []      -> accumulator
    | x :: xs -> fold folder (folder accumulator 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
```

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

Questions?

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

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
```

Appendix
(More material)

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)
```

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 (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
```

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
```