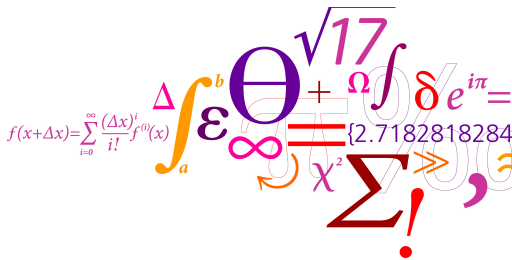


02157 Functional Programming

Disjoint Unions and Higher-order list functions

Michael R. Hansen



DTU Compute

Department of Applied Mathematics and Computer Science

- About F#
- Disjoint union (or Tagged Values)
 - Groups different kinds of values into a single set.
- Higher-order functions on lists
 - many list functions follow “standard” schemes
 - avoid (almost) identical code fragments by parameterizing functions with functions
 - higher-order list functions based on natural concepts
 - succinct declarations achievable using higher-order functions
- On types for functions briefly

Precedence and associativity rules for expressions

Operator	Association	Precedence
<code>**</code>	Associates to the right	highest
<code>* / %</code>	Associates to the left	
<code>+ -</code>	Associates to the left	
<code>= <> > >= < <=</code>	No association	
<code>& &</code>	Associates to the left	
<code> </code>	Associates to the left	lowest

- a monadic operator has higher precedence than any dyadic
- higher (larger) precedence means earlier evaluation

For example:

- `- 2 - 5 * 7 > 3 - 1` means $((-2) - (5 * 7)) > (3 - 1)$
- `fact 2 - 4` means $(\text{fact } 2) - 4$

Precedence and associativity rules for expressions

Operator	Association	Precedence
<code>**</code>	Associates to the right	highest
<code>* / %</code>	Associates to the left	
<code>+ -</code>	Associates to the left	
<code>= <> > >= < <=</code>	No association	
<code>& &</code>	Associates to the left	
<code> </code>	Associates to the left	lowest

- a monadic operator has higher precedence than any dyadic
- higher (larger) precedence means earlier evaluation
- function application associates to the left

For example:

- `- 2 - 5 * 7 > 3 - 1` means $((-2) - (5 * 7)) > (3 - 1)$
- `fact 2 - 4` means $(\text{fact } 2) - 4$
- `e1 e2 e3 e4` means $((e_1 \ e_2) \ e_3) \ e_4$

Precedence and associativity rules for expressions

Operator	Association	Precedence
<code>**</code>	Associates to the right	highest
<code>* / %</code>	Associates to the left	
<code>+ -</code>	Associates to the left	
<code>= <> > >= < <=</code>	No association	
<code>& &</code>	Associates to the left	
<code> </code>	Associates to the left	lowest

- a monadic operator has higher precedence than any dyadic
- higher (larger) precedence means earlier evaluation
- function application associates to the left
- abstraction `fun x -> e` extends as far to the right as possible

For example:

- `- 2 - 5 * 7 > 3 - 1` means `((-2)-(5*7)) > (3-1)`
- `fact 2 - 4` means `(fact 2) - 4`
- `e1 e2 e3 e4` means `((e1 e2) e3) e4`
- `fun x -> x 2` means `????`

Precedence and associativity rules for types

- infix type operators: $*$ and \rightarrow
- suffix type operator: `list`

Precedence and associativity rules for types

- infix type operators: $*$ and \rightarrow
- suffix type operator: `list`

Association rules:

- $*$ has NO association
- \rightarrow associates to the right

Precedence and associativity rules for types

- infix type operators: `*` and `->`
- suffix type operator: `list`

Association rules:

- `*` has NO association
- `->` associates to the right

Precedence rules:

- The suffix operator `list` has highest precedence
- `*` has higher precedence than `->`

For example:

Precedence and associativity rules for types

- infix type operators: `*` and `->`
- suffix type operator: `list`

Association rules:

- `*` has NO association
- `->` associates to the right

Precedence rules:

- The suffix operator `list` has highest precedence
- `*` has higher precedence than `->`

For example:

- `int*int*int list` means `(int*int*(int list))`

Precedence and associativity rules for types

- infix type operators: `*` and `->`
- suffix type operator: `list`

Association rules:

- `*` has NO association
- `->` associates to the right

Precedence rules:

- The suffix operator `list` has highest precedence
- `*` has higher precedence than `->`

For example:

- `int*int*int list` means `(int*int*(int list))`
- `int->int->int->int` means `int->(int->(int->int))`

Precedence and associativity rules for types

- infix type operators: `*` and `->`
- suffix type operator: `list`

Association rules:

- `*` has NO association
- `->` associates to the right

Precedence rules:

- The suffix operator `list` has highest precedence
- `*` has higher precedence than `->`

For example:

- `int*int*int list` means `(int*int*(int list))`
- `int->int->int->int` means `int->(int->(int->int))`
- `'a*'b->'a*'b list` means `('a*'b)->('a*('b list))`

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...

Overview: Syntactical constructs in “our part of” F#

- Constants: 0 , 1.1 , true , ...
- Patterns: x $-$ (p_1, \dots, p_n) $p_1 :: p_2$ $[p_1; \dots; p_n]$
 $p_1 | p_2$ $p \text{ when } e$ $p \text{ as } x$ $p : t \dots$

Overview: Syntactical constructs in “our part of” F#

- Constants: $0, 1.1, \text{true}, \dots$
- Patterns: $x _ (p_1, \dots, p_n) \ p_1 :: p_2 \ [p_1; \dots; p_n]$
 $p_1 | p_2 \ p \text{ when } e \ p \text{ as } x \ p :: t \dots$
- Expressions: $x \ (e_1, \dots, e_n) \ e_1 :: e_2 \ [p_1; \dots; p_n]$
 $e_1 e_2 \ e_1 \oplus e_2 \ (\oplus) \ \text{let } p_1 = e_1 \text{ in } e_2$
 $e :: t \ \text{if } e \text{ then } e_1 \text{ then } e_2 \ \text{match } e \text{ with } \textit{clauses}$
 $\text{fun } p_1 \dots p_n \rightarrow e \ \text{function } \textit{clauses} \ \dots$

where the construct *clauses* has the form:

$$| \ p_1 \rightarrow e_1 \ | \ \dots \ | \ p_n \rightarrow e_n$$

Overview: Syntactical constructs in “our part of” F#

- Constants: $0, 1.1, \text{true}, \dots$
- Patterns: $x _ (p_1, \dots, p_n) \ p_1 :: p_2 \ [p_1; \dots; p_n]$
 $p_1 | p_2 \ p \text{ when } e \ p \text{ as } x \ p : t \dots$
- Expressions: $x \ (e_1, \dots, e_n) \ e_1 :: e_2 \ [p_1; \dots; p_n]$
 $e_1 e_2 \ e_1 \oplus e_2 \ (\oplus) \ \text{let } p_1 = e_1 \text{ in } e_2$
 $e : t \ \text{if } e \text{ then } e_1 \text{ then } e_2 \ \text{match } e \text{ with } \textit{clauses}$
 $\text{fun } p_1 \dots p_n \rightarrow e \ \text{function } \textit{clauses} \ \dots$
- Declarations $\text{let } f \ p_1 \dots p_n = e \ \text{let rec } f \ p_1 \dots p_n = e, n \geq 0$

where the construct *clauses* has the form:

$$| \ p_1 \rightarrow e_1 \ | \ \dots \ | \ p_n \rightarrow e_n$$

Overview: Syntactical constructs in “our part of” F#

- Constants: `0`, `1.1`, `true`, ...
- Patterns: `x` `_` `(p1, ..., pn)` `p1::p2` `[p1; ...; pn]`
`p1|p2` `p when e` `p as x` `p:t...`
- Expressions: `x` `(e1, ..., en)` `e1::e2` `[p1; ...; pn]`
`e1e2` `e1⊕e2` `(⊕)` `let p1 = e1 in e2`
`e:t` `if e then e1 then e2` `match e with clauses`
`fun p1 ... pn -> e` `function clauses` ...
- Declarations `let f p1 ... pn = e` `let rec f p1 ... pn = e, n ≥ 0`
- Types
`int` `float` `bool` `string` `'a` `T<t1, ..., tn>` ...
`t1*t2*...*tn` `t list` `t1->t2` ...

where the construct *clauses* has the form:

`| p1 -> e1 | ... | pn -> en`

Overview: Syntactical constructs in “our part of” F#

- Constants: `0`, `1.1`, `true`, ...
- Patterns: `x` `_` `(p1, ..., pn)` `p1::p2` `[p1; ...; pn]`
`p1|p2` `p when e` `p as x` `p:t...`
- Expressions: `x` `(e1, ..., en)` `e1::e2` `[p1; ...; pn]`
`e1e2` `e1⊕e2` `(⊕)` `let p1 = e1 in e2`
`e:t` `if e then e1 then e2` `match e with clauses`
`fun p1 ... pn -> e` `function clauses` ...
- Declarations `let f p1 ... pn = e` `let rec f p1 ... pn = e`, $n \geq 0$
- Types
`int` `float` `bool` `string` `'a` `T<t1, ..., tn>` ...
`t1*t2*...*tn` `t list` `t1->t2` ...
- Type abbreviations `type T<'a1, ..., 'an> = t`

where the construct *clauses* has the form:

`| p1 -> e1 | ... | pn -> en`

Overview: Syntactical constructs in “our part of” F#

- Constants: `0`, `1.1`, `true`, ...
- Patterns: `x` `_` `(p1, ..., pn)` `p1 :: p2` `[p1; ...; pn]`
`p1 | p2` `p when e` `p as x` `p : t ...`
- Expressions: `x` `(e1, ..., en)` `e1 :: e2` `[p1; ...; pn]`
`e1 e2` `e1 ⊕ e2` `(⊕)` `let p1 = e1 in e2`
`e : t` `if e then e1 then e2` `match e with clauses`
`fun p1 ... pn -> e` `function clauses` ...
- Declarations `let f p1 ... pn = e` `let rec f p1 ... pn = e`, $n \geq 0$
- Types
`int` `float` `bool` `string` `'a` `T < t1, ..., tn >` ...
`t1 * t2 * ... * tn` `t list` `t1 -> t2` ...
- Type abbreviations `type T < 'a1, ..., 'an > = t`
- Type declarations `type T < 'a1, ..., 'an > = C1 | ... | Ci of ti | ...`

where the construct *clauses* has the form:

`| p1 -> e1 | ... | pn -> en`

Disjoint Sets – Tagged Values

Part I: Disjoint Sets – An Example

A *shape* is either a *circle*, a *square*, or a *triangle*

- the union of *three disjoint* sets

Part I: Disjoint Sets – An Example

A *shape* is either a *circle*, a *square*, or a *triangle*

- the union of *three disjoint* sets

```
type Shape =  
  | Circle of float           (* 1 *)  
  | Square of float           (* 2 *)  
  | Triangle of float*float*float;; (* 3 *)
```

Part I: Disjoint Sets – An Example

A *shape* is either a *circle*, a *square*, or a *triangle*

- the union of *three disjoint* sets

```
type Shape =
  | Circle of float                (* 1 *)
  | Square of float                (* 2 *)
  | Triangle of float*float*float;; (* 3 *)
```

This declaration provides three *rules* for generating *shapes*:

- if $r : \text{float}$, then $\text{Circle } r : \text{Shape}$ (* 1 *)
- if $s : \text{float}$, then $\text{Square } s : \text{Shape}$ (* 2 *)
- if $(a, b, c) : \text{float} * \text{float} * \text{float}$,
then $\text{Triangle}(a, b, c) : \text{Shape}$ (* 3 *)

Part I: Disjoint Sets – An Example

A *shape* is either a *circle*, a *square*, or a *triangle*

- the union of *three disjoint* sets

```
type Shape =
  | Circle of float                (* 1 *)
  | Square of float                (* 2 *)
  | Triangle of float*float*float;; (* 3 *)
```

This declaration provides three *rules* for generating *shapes*:

- if *r* : float, then Circle *r* : Shape (* 1 *)
- if *s* : float, then Square *s* : Shape (* 2 *)
- if (*a*, *b*, *c*) : float*float*float,
then Triangle(*a*, *b*, *c*) : Shape (* 3 *)

A type like *Shape* is also called an *algebraic data type*

Part I: Disjoint Sets – An Example (II)

The tags `Circle`, `Square` and `Triangle` are called **constructors**:

```
Circle    : float → Shape  
Square    : float → Shape  
Triangle  : float * float * float → Shape
```


Part I: Disjoint Sets – An Example (II)

The tags `Circle`, `Square` and `Triangle` are called **constructors**:

```
Circle    : float → Shape
Square    : float → Shape
Triangle  : float * float * float → Shape
```

```
- Circle 2.0;;
> val it : Shape = Circle 2.0

- Triangle(1.0, 2.0, 3.0);;
> val it : Shape = Triangle(1.0, 2.0, 3.0)

- Square 4.0;;
> val it : Shape = Square 4.0
```

Part I: Disjoint Sets – An Example (II)

The tags `Circle`, `Square` and `Triangle` are called **constructors**:

```
Circle    : float → Shape
Square    : float → Shape
Triangle  : float * float * float → Shape
```

```
- Circle 2.0;;
> val it : Shape = Circle 2.0

- Triangle(1.0, 2.0, 3.0);;
> val it : Shape = Triangle(1.0, 2.0, 3.0)

- Square 4.0;;
> val it : Shape = Square 4.0
```

`Circle`, `Square` and `Triangle` are used
to construct values of type **Shape**,

A shape-area function is declared

```
let area =  
  function  
  | Circle r          -> System.Math.PI * r * r  
  | Square a          -> a * a  
  | Triangle(a,b,c) ->  
    let s = (a + b + c)/2.0  
    sqrt(s*(s-a)*(s-b)*(s-c));;  
> val area : Shape -> float
```

following the structure of shapes.

using Heron's formula

A shape-area function is declared

```
let area =  
  function  
  | Circle r          -> System.Math.PI * r * r  
  | Square a          -> a * a  
  | Triangle(a,b,c) ->  
    let s = (a + b + c)/2.0  
    sqrt(s*(s-a)*(s-b)*(s-c));;  
> val area : Shape -> float
```

following the structure of shapes.

using Heron's formula

- a constructor only matches itself

Constructors in Patterns

A shape-area function is declared

```
let area =
  function
  | Circle r          -> System.Math.PI * r * r
  | Square a          -> a * a
  | Triangle(a,b,c) ->
      let s = (a + b + c)/2.0
      sqrt(s*(s-a)*(s-b)*(s-c));;
> val area : Shape -> float
```

following the structure of shapes.

using Heron's formula

- a constructor only matches itself

```
area (Circle 1.2)
~> (System.Math.PI * r * r, [r ↦ 1.2])
~> ...
```

Months are naturally defined using tagged values::

```
type Month = | January | February | March | April  
            | May | June | July | August | September  
            | October | November | December;;
```

Months are naturally defined using tagged values::

```
type Month = | January | February | March | April  
            | May | June | July | August | September  
            | October | November | December;;
```

The days-in-a-month function is declared by

```
let daysOfMonth =  
  function  
    | February                -> 28  
    | April | June | September | November -> 30  
    | _                      -> 31;;  
val daysOfMonth : Month -> int
```

Enumeration types – the months

Months are naturally defined using tagged values::

```
type Month = | January | February | March | April  
            | May | June | July | August | September  
            | October | November | December;;
```

The days-in-a-month function is declared by

```
let daysOfMonth =  
  function  
    | February                -> 28  
    | April | June | September | November -> 30  
    | _                      -> 31;;  
val daysOfMonth : Month -> int
```

Observe: Constructors need not have arguments

The `option` type

```
type 'a option = None | Some of 'a
```

Distinguishes the cases "nothing" and "something".

`predefined`

The `option` type

```
type 'a option = None | Some of 'a
```

Distinguishes the cases "nothing" and "something".

predefined

The constructor `Some` and `None` are polymorphic:

```
Some false;;
```

```
val it : bool option = Some false
```

```
Some (1, "a");;
```

```
val it : (int * string) option = Some (1, "a")
```

```
None;;
```

```
val it : 'a option = None
```

The `option` type

```
type 'a option = None | Some of 'a
```

Distinguishes the cases "nothing" and "something".

predefined

The constructor `Some` and `None` are polymorphic:

```
Some false;;  
val it : bool option = Some false
```

```
Some (1, "a");;  
val it : (int * string) option = Some (1, "a")
```

```
None;;  
val it : 'a option = None
```

Observe: type variables are allowed in declarations of algebraic types

Example: Find first position of element in a list

```
let rec findPosA p x =  
  function  
  | y::_ when x=y -> Some p  
  | _::ys         -> findPosA (p+1) x ys  
  | []           -> None;;  
val findPosA : int -> 'a -> 'a list -> int option when ...
```

Example: Find first position of element in a list

```
let rec findPosA p x =  
  function  
  | y::_ when x=y -> Some p  
  | _::ys          -> findPosA (p+1) x ys  
  | []             -> None;;  
val findPosA : int -> 'a -> 'a list -> int option when ...  
  
let findPos x ys = findPosA 0 x ys;;  
val findPos : 'a -> 'a list -> int option when ...
```

Example: Find first position of element in a list

```
let rec findPosA p x =  
  function  
  | y::_ when x=y -> Some p  
  | _::ys          -> findPosA (p+1) x ys  
  | []             -> None;;  
val findPosA : int -> 'a -> 'a list -> int option when ...  
  
let findPos x ys = findPosA 0 x ys;;  
val findPos : 'a -> 'a list -> int option when ...
```

Examples

```
findPos 4 [2 .. 6];;  
val it : int option = Some 2
```

```
findPos 7 [2 .. 6];;  
val it : int option = None
```

```
Option.get(findPos 4 [2 .. 6]);;  
val it : int = 2
```

Declaration of Algebraic data types yield **new** types

```
type T = A of int;;
```

```
let v = A 1;;
```

```
val v: T = A 1
```

Declaration of Algebraic data types yield **new** types

```
type T = A of int;;
```

```
let v = A 1;;
```

```
val v: T = A 1
```

```
type T = A of int;;           // declaration of a new type
```

```
let v' = A 1;;
```

```
val v': T = A 1
```

```
v=v';;
```


Declaration of Algebraic data types yield **new** types

```
type T = A of int;;
```

```
let v = A 1;;  
val v: T = A 1
```

```
type T = A of int;;           // declaration of a new type
```

```
let v' = A 1;;  
val v': T = A 1
```

```
v=v';;
```

```
error FS0001: This expression was expected to have  
type 'FSI_0011.T' but here has type 'FSI_0013.T'
```

The second declaration do indeed yield a **genuine new type**.

A (teaching) room at DTU is either an auditorium or a databar:

- an auditorium is characterized by a location and a number of seats.
- a databar is characterized by a location, a number of computers and a number of seats.

Declare a type *Room*.

A (teaching) room at DTU is either an auditorium or a databar:

- an auditorium is characterized by a location and a number of seats.
- a databar is characterized by a location, a number of computers and a number of seats.

Declare a type *Room*.

Declare a function:

seatCapacity : Room \rightarrow int

A (teaching) room at DTU is either an auditorium or a databar:

- an auditorium is characterized by a location and a number of seats.
- a databar is characterized by a location, a number of computers and a number of seats.

Declare a type *Room*.

Declare a function:

seatCapacity : Room \rightarrow int

Declare a function

computerCapacity : Room \rightarrow int option

Declaration of monomorphic and polymorphic types

A declaration of a **monomorphic** type has the form:

`type $T = t$`

where t **does not contain type variables**.

Declaration of monomorphic and polymorphic types

A declaration of a **monomorphic** type has the form:

$$\text{type } T = t$$

where t **does not contain type variables**.

A declaration of a **polymorphic type** has the form:

$$\text{type } T <'a_0, 'a_1, \dots, 'a_n> = t$$

where t may **contain type variables** $'a_0, 'a_1, \dots, 'a_n$.

Declaration of monomorphic and polymorphic types

A declaration of a **monomorphic** type has the form:

$$\text{type } T = t$$

where t **does not contain type variables**.

A declaration of a **polymorphic type** has the form:

$$\text{type } T <'a_0, 'a_1, \dots, 'a_n> = t$$

where t may **contain type variables** $'a_0, 'a_1, \dots, 'a_n$.

A declaration of a polymorphic type

- $\text{type Map} <'c> = ('c * 'c) \text{ list}$

Declaration of monomorphic and polymorphic types

A declaration of a **monomorphic** type has the form:

$$\text{type } T = t$$

where t **does not contain type variables**.

A declaration of a **polymorphic type** has the form:

$$\text{type } T <'a_0, 'a_1, \dots, 'a_n> = t$$

where t may **contain type variables** $'a_0, 'a_1, \dots, 'a_n$.

A declaration of a polymorphic type

- `type Map<'c> = ('c * 'c) list`

Declarations of monomorphic types:

- `type Country = A | B | C | D | E | F`
- `type SmallMap = Map<Country>`

Higher-order list functions

Higher-order functions are

- everywhere

$$\sum_{i=a}^b f(i), \frac{df}{dx}, \{x \in A \mid P(x)\}, \dots$$

Higher-order functions are

- everywhere

$$\sum_{i=a}^b f(i), \frac{df}{dx}, \{x \in A \mid P(x)\}, \dots$$

- powerful

Parameterized modules, succinct code ...

Higher-order functions are

- everywhere

$$\sum_{i=a}^b f(i), \frac{df}{dx}, \{x \in A \mid P(x)\}, \dots$$

- powerful

Parameterized modules, succinct code ...

HIGHER-ORDER FUNCTIONS ARE USEFUL

now down to earth

now down to earth

- Many recursive declarations follows the same schema.

For example:

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

now down to earth

- Many recursive declarations follows the same schema.

For example:

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

Succinct declarations achievable using higher-order functions

Contents

- Higher-order list functions (in the library)
 - map
 - contains, exists, forall, filter, tryFind
 - foldBack, fold

now down to earth

- Many recursive declarations follows the same schema.

For example:

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

Succinct declarations achievable using higher-order functions

Contents

- Higher-order list functions (in the library)
 - map
 - contains, exists, forall, filter, tryFind
 - foldBack, fold

Avoid (almost) identical code fragments by
parameterizing functions with functions

A simple declaration of a list function

A typical declaration following the structure of lists:

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

posList [4; -5; 6];;
val it : bool list = [true; false; true]
```

A simple declaration of a list function

A typical declaration following the structure of lists:

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

posList [4; -5; 6];;
val it : bool list = [true; false; true]
```

Applies the function `fun x -> x > 0` to each element in a list

Another declaration with the same structure

```
let rec addElems = function
    | []          -> []
    | (x,y)::zs -> (x+y)::addElems zs;;
val addElems : (int * int) list -> int list

addElems [(1,2) ; (3,4)];;
val it : int list = [3; 7]
```

Another declaration with the same structure

```
let rec addElems = function
    | []          -> []
    | (x,y)::zs -> (x+y)::addElems zs;;
val addElems : (int * int) list -> int list

addElems [(1,2) ; (3,4)];;
val it : int list = [3; 7]
```

Applies the addition function + to each pair of integers in a list

The function: `map`

Applies a function to each element in a list

$$\text{map } f [v_1; v_2; \dots; v_n] = [f(v_1); f(v_2); \dots; f(v_n)]$$

Declaration

Library function

```
let rec map f = function
  | []      -> []
  | x::xs   -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

The function: `map`

Applies a function to each element in a list

$$\text{map } f [v_1; v_2; \dots; v_n] = [f(v_1); f(v_2); \dots; f(v_n)]$$

Declaration

Library function

```
let rec map f = function
    | []      -> []
    | x::xs   -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

Succinct declarations can be achieved using `map`, e.g.

```
let posList = map (fun x -> x > 0);;
val posList : int list -> bool list

let addElems = map (fun (x,y) -> x+y);;
val addElems : (int * int) list -> int list
```

Declare a function

$$g [x_1, \dots, x_n] = [x_1^2 + 1, \dots, x_n^2 + 1]$$

Remember

$$\text{map } f [v_1; v_2; \dots; v_n] = [f(v_1); f(v_2); \dots; f(v_n)]$$

where

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

Higher-order list functions: `exists`

Predicate: For some x in xs : $p(x)$.

Higher-order list functions: `exists`

Predicate: For some x in xs : $p(x)$.

$$\text{exists } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `exists`

Predicate: For some x in xs : $p(x)$.

$$\text{exists } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

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

Higher-order list functions: `exists`

Predicate: For some x in xs : $p(x)$.

$$\text{exists } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

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

Example

```
exists (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = true
```

Declare `contains` function using `exists`.

```
let contains x ys = exists ????? ;;  
val contains : 'a -> 'a list -> bool when 'a : equality
```

Remember

$$\text{exists } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

where

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

`contains` is a Library function

Higher-order list functions: `forall`

Predicate: For every x in xs : $p(x)$.

Higher-order list functions: `forall`

Predicate: For every x in xs : $p(x)$.

$$\text{forall } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `forall`

Predicate: For every x in xs : $p(x)$.

$$\text{forall } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec forall p = function
    | []      -> true
    | x::xs -> p x && forall p xs;;
val forall : ('a -> bool) -> 'a list -> bool
```

Higher-order list functions: `forall`

Predicate: For every x in xs : $p(x)$.

$$\text{forall } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec forall p = function
  | []      -> true
  | x::xs -> p x && forall p xs;;
val forall : ('a -> bool) -> 'a list -> bool
```

Example

```
forall (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = false
```


Declare a function

`disjoint xs ys`

which is true when there are no common elements in the lists `xs` and `ys`, and false otherwise.

Remember

$$\text{forall } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

where

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

Validating properties of map-coloring programs (I)

Remember:

Script available on Learn

```
type Map<'c>      = ('c * 'c) list
type Color<'c>    = 'c list
type Coloring<'c> = Color<'c> list
```

An auxiliary function in the script:

```
countries: Map<'c> -> 'c list
```

should give a list containing all elements in a map.

Validating properties of map-coloring programs (I)

Remember:

Script available on Learn

```
type Map<'c>      = ('c * 'c) list
type Color<'c>    = 'c list
type Coloring<'c> = Color<'c> list
```

An auxiliary function in the script:

```
countries: Map<'c> -> 'c list
```

should give a list containing all elements in a map.

The function should satisfy the property $\text{prop}_1(m)$:
every country in map m must be in `countries m`.

Validating properties of map-coloring programs (I)

Remember:

Script available on Learn

```
type Map<'c>      = ('c * 'c) list
type Color<'c>    = 'c list
type Coloring<'c> = Color<'c> list
```

An auxiliary function in the script:

```
countries: Map<'c> -> 'c list
```

should give a list containing all elements in a map.

The function should satisfy the property $\text{prop}_1(m)$:
every country in map m must be in `countries m`.

How to declare a function that can check this property?

Validating properties of map-coloring programs (II)

We restrict our attention to a small number of countries when using
FsCheck to validate `prop1`. **Why?**

Validating properties of map-coloring programs (II)

We restrict our attention to a small number of countries when using FsCheck to validate `prop1`. Why?

```
#r "nuget: FsCheck";;
open FsCheck;;

type Country = A | B | C | D | E | F
type SmallMap = Map<Country>

let prop1(m:SmallMap) =
    let cs = countries m
    List.forall (fun (c1,c2) -> List.contains c1 cs &&
                                List.contains c2 cs ) m;;

let _ = Check.Verbose prop1;;
...
9:
[(A, B); (D, B); (D, A)]
...
Ok, passed 100 tests.
```

Validating properties of map-coloring programs (II)

We restrict our attention to a small number of countries when using FsCheck to validate `prop1`. Why?

```
#r "nuget: FsCheck";;
open FsCheck;;

type Country = A | B | C | D | E | F
type SmallMap = Map<Country>

let prop1(m:SmallMap) =
  let cs = countries m
  List.forall (fun (c1,c2) -> List.contains c1 cs &&
                                List.contains c2 cs ) m;;

let _ = Check.Verbose prop1;;
...
9:
[(A, B); (D, B); (D, A)]
...
Ok, passed 100 tests.
```

Properties should be monomorphic

Higher-order list functions: `filter`

Set comprehension: $\{x \in xs : p(x)\}$

Higher-order list functions: `filter`

Set comprehension: $\{x \in xs : p(x)\}$

`filter p xs` is the list of those elements x of xs where $p(x) = \text{true}$.

Higher-order list functions: `filter`

Set comprehension: $\{x \in xs : p(x)\}$

`filter p xs` is the list of those elements x of xs where $p(x) = \text{true}$.

Declaration

Library function

```
let rec filter p =  
  function  
  | []      -> []  
  | x::xs -> if p x then x :: filter p xs  
              else filter p xs;;  
val filter : ('a -> bool) -> 'a list -> 'a list
```

Higher-order list functions: `filter`

Set comprehension: $\{x \in xs : p(x)\}$

`filter p xs` is the list of those elements x of xs where $p(x) = \text{true}$.

Declaration

Library function

```
let rec filter p =
  function
  | []      -> []
  | x::xs -> if p x then x :: filter p xs
              else filter p xs;;
val filter : ('a -> bool) -> 'a list -> 'a list
```

Example

```
filter System.Char.IsLetter ['l'; 'p'; 'F'; '-'];;
val it : char list = ['p'; 'F']
```

where `System.Char.IsLetter c` is true iff
 $c \in \{'A', \dots, 'Z'\} \cup \{'a', \dots, 'z'\}$

Declare a function

```
inter xs ys
```

which contains the common elements of the lists *xs* and *ys* — i.e. their intersection.

Order and repetition of elements are of no concern

Remember:

`filter p xs` is the list of those elements *x* of *xs* where *p(x) = true*. where

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

Higher-order list functions: `tryFind`

$$\text{tryFind } p \text{ } xs = \begin{cases} \text{Some } x & \text{for an element } x \text{ of } xs \text{ with } p(x) = \text{true} \\ \text{None} & \text{if no such element exists} \end{cases}$$

Higher-order list functions: `tryFind`
$$\text{tryFind } p \text{ } xs = \begin{cases} \text{Some } x & \text{for an element } x \text{ of } xs \text{ with } p(x) = \text{true} \\ \text{None} & \text{if no such element exists} \end{cases}$$

```
let rec tryFind p =  
  function  
  | x::xs when p x -> Some x  
  | _::xs      -> tryFind p xs  
  | _         -> None;;  
val tryFind : ('a -> bool) -> 'a list -> 'a option
```

Higher-order list functions: `tryFind`
$$\text{tryFind } p \text{ } xs = \begin{cases} \text{Some } x & \text{for an element } x \text{ of } xs \text{ with } p(x) = \text{true} \\ \text{None} & \text{if no such element exists} \end{cases}$$

```
let rec tryFind p =  
  function  
  | x::xs when p x -> Some x  
  | _::xs      -> tryFind p xs  
  | _         -> None;;  
val tryFind : ('a -> bool) -> 'a list -> 'a option
```

```
tryFind (fun x -> x>3) [1;5;-2;8];;  
val it : int option = Some 5
```

Folding a function over a list (I)

Example: sum of absolute values:

```
let rec absSum = function
    | []      -> 0
    | x::xs  -> abs x + absSum xs;;
val absSum : int list -> int

absSum [-2; 2; -1; 1; 0];;
val it : int = 6
```


Folding a function over a list (II)

```
let rec absSum = function
  | []      -> 0
  | x::xs -> abs x + absSum xs;;
```

Let $f\ x\ a$ abbreviate $\text{abs } x + a$ in the evaluation:

$$\begin{aligned}
 & \text{absSum } [x_0; x_1; \dots; x_{n-1}] \\
 \rightsquigarrow & \text{abs } x_0 + (\text{absSum } [x_1; \dots; x_{n-1}]) \\
 = & f\ x_0 (\text{absSum } [x_1; \dots; x_{n-1}]) \\
 \rightsquigarrow & f\ x_0 (f\ x_1 (\text{absSum } [x_2; \dots; x_{n-1}])) \\
 & \vdots \\
 \rightsquigarrow & f\ x_0 (f\ x_1 (\dots (f\ x_{n-1} 0) \dots))
 \end{aligned}$$

Folding a function over a list (II)

```
let rec absSum = function
  | []      -> 0
  | x::xs -> abs x + absSum xs;;
```

Let $f\ x\ a$ abbreviate $\text{abs } x + a$ in the evaluation:

$$\begin{aligned}
 & \text{absSum } [x_0; x_1; \dots; x_{n-1}] \\
 \rightsquigarrow & \text{abs } x_0 + (\text{absSum } [x_1; \dots; x_{n-1}]) \\
 = & f\ x_0 (\text{absSum } [x_1; \dots; x_{n-1}]) \\
 \rightsquigarrow & f\ x_0 (f\ x_1 (\text{absSum } [x_2; \dots; x_{n-1}])) \\
 & \vdots \\
 \rightsquigarrow & f\ x_0 (f\ x_1 (\dots (f\ x_{n-1} 0) \dots))
 \end{aligned}$$

This repeated application of f is also called a **folding** of f .

Folding a function over a list (II)

```
let rec absSum = function
  | []      -> 0
  | x::xs -> abs x + absSum xs;;
```

Let $f\ x\ a$ abbreviate $\text{abs } x + a$ in the evaluation:

$$\begin{aligned}
 & \text{absSum } [x_0; x_1; \dots; x_{n-1}] \\
 \rightsquigarrow & \text{abs } x_0 + (\text{absSum } [x_1; \dots; x_{n-1}]) \\
 = & f\ x_0 (\text{absSum } [x_1; \dots; x_{n-1}]) \\
 \rightsquigarrow & f\ x_0 (f\ x_1 (\text{absSum } [x_2; \dots; x_{n-1}])) \\
 & \vdots \\
 \rightsquigarrow & f\ x_0 (f\ x_1 (\dots (f\ x_{n-1} 0) \dots))
 \end{aligned}$$

This repeated application of f is also called a **folding** of f .

Many functions follow such recursion and evaluation schemes

Higher-order list functions: `foldBack` (1)

Suppose that \otimes is an infix function. Then

$$\begin{aligned} \text{foldBack } (\otimes) \ [a_0; a_1; \dots; a_{n-2}; a_{n-1}] \ e_b \\ = \ a_0 \otimes (a_1 \otimes (\dots (a_{n-2} \otimes (a_{n-1} \otimes e_b)) \dots)) \end{aligned}$$

Higher-order list functions: `foldBack` (1)

Suppose that \otimes is an infix function. Then

$$\begin{aligned} \text{foldBack } (\otimes) \ [a_0; a_1; \dots; a_{n-2}; a_{n-1}] \ e_b \\ = \ a_0 \otimes (a_1 \otimes (\dots (a_{n-2} \otimes (a_{n-1} \otimes e_b)) \dots)) \end{aligned}$$

$$\text{List.foldBack } (+) \ [1; 2; 3] \ 0 \ = \ 1 + (2 + (3 + 0)) \ = \ 6$$

$$\text{List.foldBack } (-) \ [1; 2; 3] \ 0 \ = \ 1 - (2 - (3 - 0)) \ = \ 2$$

Declaration of `foldBack`



```
let rec foldBack f xlst e =  
  match xlst with  
  | x::xs -> f x (foldBack f xs e)  
  | []      -> e;;  
val foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
let rec foldBack f xlst e =  
  match xlst with  
  | x::xs -> f x (foldBack f xs e)  
  | []      -> e;;  
val foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
let absSum xs = foldBack (fun x a -> abs x + a) xs 0;;
```

```
let length xs = foldBack (fun _ n -> n+1) xs 0;;
```

```
let map f xs = foldBack (fun x rs -> f x :: rs) xs [];;
```

Let an insertion function be declared by

```
let insert x ys = if List.contains x ys then ys
                  else x::ys;;
```

Declare a function `distinct xs`, that returns a list having same elements as `xs` but without duplicated element.

Remember:

$$\text{foldBack } (\oplus) [x_1; x_2; \dots; x_n] b \rightsquigarrow x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus b) \dots)$$

where

```
foldBack: ('a -> 'state -> 'state)
          -> 'a list -> 'state -> 'state
```


Higher-order list functions: `fold` (1)

Suppose that \oplus is an infix function.

Then the `fold` function is defined by:

$$\begin{aligned} \text{fold } (\oplus) \ e_a \ [b_0; b_1; \dots; b_{n-2}; b_{n-1}] \\ = \ ((\dots((e_a \oplus b_0) \oplus b_1) \dots) \oplus b_{n-2}) \oplus b_{n-1} \end{aligned}$$

i.e. it applies \oplus from left to right.

Higher-order list functions: `fold` (1)

Suppose that \oplus is an infix function.

Then the `fold` function is defined by:

$$\begin{aligned} \text{fold } (\oplus) e_a [b_0; b_1; \dots; b_{n-2}; b_{n-1}] \\ = ((\dots((e_a \oplus b_0) \oplus b_1) \dots) \oplus b_{n-2}) \oplus b_{n-1} \end{aligned}$$

i.e. it applies \oplus from left to right.

Examples:

$$\text{List.fold } (-) 0 [1; 2; 3] = ((0 - 1) - 2) - 3 = -6$$

Higher-order list functions: `fold` (1)

Suppose that \oplus is an infix function.

Then the `fold` function is defined by:

$$\begin{aligned} \text{fold } (\oplus) \ e_a \ [b_0; b_1; \dots; b_{n-2}; b_{n-1}] \\ = \ ((\dots((e_a \oplus b_0) \oplus b_1) \dots) \oplus b_{n-2}) \oplus b_{n-1} \end{aligned}$$

i.e. it applies \oplus from left to right.

Examples:

$$\begin{aligned} \text{List.fold } (-) \ 0 \ [1; 2; 3] &= ((0 - 1) - 2) - 3 = -6 \\ \text{List.foldBack } (-) \ [1; 2; 3] \ 0 &= 1 - (2 - (3 - 0)) = 2 \end{aligned}$$

Higher-order list functions: `fold` (2)

```
let rec fold f e =  
  function  
  | x::xs -> fold f (f e x) xs  
  | []     -> e;;  
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Higher-order list functions: `fold` (2)

```
let rec fold f e =  
  function  
  | x::xs -> fold f (f e x) xs  
  | []     -> e;;  
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Using `cons` in connection with `fold` gives the reverse function:

```
let rev xs = fold (fun rs x -> x::rs) [] xs;;
```

This function has a linear execution time:

Higher-order list functions: `fold` (2)

```
let rec fold f e =
  function
  | x::xs -> fold f (f e x) xs
  | []     -> e;;
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Using cons in connection with `fold` gives the reverse function:

```
let rev xs = fold (fun rs x -> x::rs) [] xs;;
```

This function has a linear execution time:

```
rev [1;2;3]
~> fold (fun ...) [] [1;2;3]
~> fold (fun ...) (1::[]) [2;3]
~> fold (fun ...) [1] [2;3]
~> fold (fun ...) (2::[1]) [3]
~> fold (fun ...) [2;1] [3]
~> fold (fun ...) (3::[2;1]) []
~> fold (fun ...) [3;2;1] []
~> [3;2;1]
```

- Part I: Disjoint union (Algebraic data types)
 - types containing different kinds of values

Part I: Disjoint union (Algebraic data types)

- types containing different kinds of values

We will later extend the notion with recursive definitions

- provides a mean to declare types for finite trees

Part I: Disjoint union (Algebraic data types)

- types containing different kinds of values

We will later extend the notion with recursive definitions

- provides a mean to declare types for finite trees

Part II: Higher-order list functions

- Many recursive declarations follows the same schema.

Succinct declarations achievable using higher-order functions

Summary

Part I: Disjoint union (Algebraic data types)

- types containing different kinds of values

We will later extend the notion with recursive definitions

- provides a mean to declare types for finite trees

Part II: Higher-order list functions

- Many recursive declarations follows the same schema.

Succinct declarations achievable using higher-order functions

- Higher-order list functions (in the library)
 - map
 - contains, exists, forall, filter, tryFind
 - foldBack, fold

Summary

Part I: Disjoint union (Algebraic data types)

- types containing different kinds of values

We will later extend the notion with recursive definitions

- provides a mean to declare types for finite trees

Part II: Higher-order list functions

- Many recursive declarations follows the same schema.

Succinct declarations achievable using higher-order functions

- Higher-order list functions (in the library)
 - map
 - contains, exists, forall, filter, tryFind
 - foldBack, fold

Avoid (almost) identical code fragments by
parameterizing functions with functions

On the design of types for functions — briefly

Some consideration

- Is partial function application envisioned?
- Conventions
- Make functions fit together
- “nice” declaration
- “safe parentheses”
- ...

Design of (Higher-order) functions

Suppose a function can take two arguments. Possible types:

$$\tau_1 * \tau_2 \rightarrow \tau$$

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau$$

$$\tau_2 * \tau_1 \rightarrow \tau$$

$$\tau_2 \rightarrow \tau_1 \rightarrow \tau$$

The number of possibilities becomes huge when there are more types and when types have a structure.

Design of (Higher-order) functions

Suppose a function can take two arguments. Possible types:

$$\tau_1 * \tau_2 \rightarrow \tau$$

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau$$

$$\tau_2 * \tau_1 \rightarrow \tau$$

$$\tau_2 \rightarrow \tau_1 \rightarrow \tau$$

The number of possibilities becomes huge when there are more types and when types have a structure.

Types of some library functions folding from right:

- $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \rightarrow 'b)$ OCAML: `fold_right`
- $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b)$ Haskell: `foldr`
- $('a * 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b)$ SML: `foldr`
- ...

Design of (Higher-order) functions

Suppose a function can take two arguments. Possible types:

$$\tau_1 * \tau_2 \rightarrow \tau$$

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau$$

$$\tau_2 * \tau_1 \rightarrow \tau$$

$$\tau_2 \rightarrow \tau_1 \rightarrow \tau$$

The number of possibilities becomes huge when there are more types and when types have a structure.

Types of some library functions folding from right:

- $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \rightarrow 'b)$ OCAML: `fold_right`
- $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b)$ Haskell: `foldr`
- $('a * 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b)$ SML: `foldr`
- ...

It does not may sense to discuss what is best

Design of (Higher-order) functions

Suppose a function can take two arguments. Possible types:

$$\tau_1 * \tau_2 \rightarrow \tau$$

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau$$

$$\tau_2 * \tau_1 \rightarrow \tau$$

$$\tau_2 \rightarrow \tau_1 \rightarrow \tau$$

The number of possibilities becomes huge when there are more types and when types have a structure.

Types of some library functions folding from right:

- $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \rightarrow 'b)$ OCAML: `fold_right`
- $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b)$ Haskell: `foldr`
- $('a * 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b)$ SML: `foldr`
- ...

It does not may sense to discuss what is best

An observation about `foldBack` (F# and OCAML):

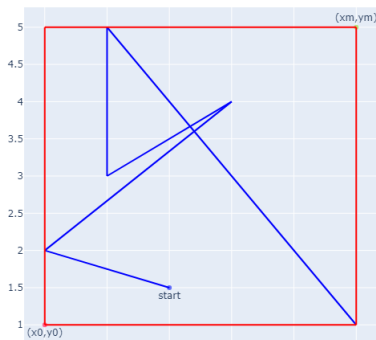
$$\begin{array}{c} \text{foldBack } (\oplus) [x_1; \dots; x_n] e \\ \rightsquigarrow x_1 \oplus (\dots (x_n \oplus e) \dots) \end{array}$$

insert \oplus between “all arguments” and evaluate from right

Piecewise linear curve with Bounding Box

```
type Point          = float * float
type Curve          = Point * Point list // (p0,[p1;...;pn])
type BoundingBox    = Point * Point      // ((x0,y0),(xm,ym))

extBB: ?             // extend a bounding box by a point
findBB: Curve -> BoundingBox
```



Computing bounding box

Could be done by repeated extension of a bounding box by a point.

Example: Extending

- bounding box $((1, 1.5), (3, 2))$
- with the point $(4, 4)$

gives the bounding box $((1, 1.5), (4, 4))$

Computing bounding box

Could be done by repeated extension of a bounding box by a point.

Example: Extending

- bounding box $((1,1.5), (3,2))$
- with the point $(4,4)$

gives the bounding box $((1,1.5), (4,4))$

Possible (natural) types:

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Computing bounding box

Could be done by repeated extension of a bounding box by a point.

Example: Extending

- bounding box $((1, 1.5), (3, 2))$
- with the point $(4, 4)$

gives the bounding box $((1, 1.5), (4, 4))$

Possible (natural) types:

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Which one should be chosen?

Computing bounding box

Could be done by repeated extension of a bounding box by a point.

Example: Extending

- bounding box $((1, 1.5), (3, 2))$
- with the point $(4, 4)$

gives the bounding box $((1, 1.5), (4, 4))$

Possible (natural) types:

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Which one should be chosen?

- if `findBB` is declared using recursion,
it is hard to have a preference

Declare `findBB` using higher-order functions (I)

The type of `List.fold`

```
fold: ('bb -> 'p -> 'bb) -> 'bb -> 'p list -> 'bb)
```

Remember

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Declare `findBB` using higher-order functions (I)

The type of `List.fold`

```
fold: ('bb -> 'p -> 'bb) -> 'bb -> 'p list -> 'bb)
```

Remember

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Four versions using `fold`:

```
let findBB1(p0,ps) =
  fold (fun bb p -> extBB1 p bb) (p0,p0) ps
```


Declare `findBB` using higher-order functions (I)

The type of `List.fold`

```
fold: ('bb -> 'p -> 'bb) -> 'bb -> 'p list -> 'bb)
```

Remember

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Four versions using `fold`:

```
let findBB1(p0,ps) =
  fold (fun bb p -> extBB1 p bb) (p0,p0) ps

let findBB2(p0,ps) =
  fold (fun bb p -> extBB2(p,bb)) (p0,p0) ps
```

Declare `findBB` using higher-order functions (I)

The type of `List.fold`

```
fold: ('bb -> 'p -> 'bb) -> 'bb -> 'p list -> 'bb)
```

Remember

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Four versions using `fold`:

```
let findBB1(p0,ps) =
  fold (fun bb p -> extBB1 p bb) (p0,p0) ps

let findBB2(p0,ps) =
  fold (fun bb p -> extBB2(p,bb)) (p0,p0) ps

let findBB3(p0,ps) = fold extBB3 (p0,p0) ps
```

Declare `findBB` using higher-order functions (I)

The type of `List.fold`

```
fold: ('bb -> 'p -> 'bb) -> 'bb -> 'p list -> 'bb)
```

Remember

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Four versions using `fold`:

```
let findBB1(p0,ps) =
  fold (fun bb p -> extBB1 p bb) (p0,p0) ps

let findBB2(p0,ps) =
  fold (fun bb p -> extBB2(p,bb)) (p0,p0) ps

let findBB3(p0,ps) = fold extBB3 (p0,p0) ps

let findBB4(p0,ps) =
  fold (fun bb p -> extBB4(bb,p)) (p0,p0) ps
```

Declare `findBB` using higher-order functions (II)

The type of `List.foldBack`

```
foldBack: ('p -> 'bb -> 'bb) -> 'p list -> 'bb -> 'bb
```

Remember

```
extBB1: Point -> BoundingBox -> BoundingBox  
extBB2: Point * BoundingBox -> BoundingBox  
extBB3: BoundingBox -> Point -> BoundingBox  
extBB4: BoundingBox * Point -> BoundingBox
```

Declare `findBB` using higher-order functions (II)

The type of `List.foldBack`

```
foldBack: ('p -> 'bb -> 'bb) -> 'p list -> 'bb -> 'bb
```

Remember

```
extBB1: Point -> BoundingBox -> BoundingBox
extBB2: Point * BoundingBox -> BoundingBox
extBB3: BoundingBox -> Point -> BoundingBox
extBB4: BoundingBox * Point -> BoundingBox
```

Just one version fits `foldBack`:

```
let findBB' (p0,ps) = foldBack extBB1 ps (p0,p0)
```

Design functions so that they **compose** nicely

Design functions so that they **compose** nicely

whatever this means

Design functions so that they **compose** nicely

whatever this means

- **extBB1** fits nicely with **foldBack**
- **extBB3** fits nicely with **fold**

Design functions so that they **compose** nicely

whatever this means

- **extBB1** fits nicely with **foldBack**
- **extBB3** fits nicely with **fold**

What about functions to move and join curves?

Let us have a look at some programs

Design functions so that they **compose** nicely

whatever this means

- **extBB1** fits nicely with **foldBack**
- **extBB3** fits nicely with **fold**

What about functions to move and join curves?

Let us have a look at some programs

Alternative type for **Curve**?