ITT8060: Advanced Programming (in F#)

Lecture 4: Tagged values / discriminated unions, Lists, Higher-order functions on lists

Juhan Ernits, Hendrik Maarand and Ian Erik Varatalu
based on slides by Michael R. Hansen

Department of Software Science, Tallinn University of Technology

27/09/2023

Recall the `match` construct

```
match test-exp with
| pattern-1 [ when c-1 ] -> result-exp-1
| ...
| pattern-n [ when c-n ] -> result-exp-n
```

Pattern matching: match (the result of evaluating) the expression `test-exp` against a series of patterns to decide what to do next.

From top to bottom.

If we have a match on a `pattern`, then also evaluate the `when` guard `c`. A missing `when` guard is the same as having `when true`.

If `c` evaluates to `true`, then the corresponding `result-exp` is the result of this `match` and we evaluate it.

If a `pattern` does not match or `c` evaluates to `false`, then continue with the next pattern.

At most one `result-exp` gets evaluated. The type of the `match` expression is determined by the `result-exp`s

Recall the `match` construct: an example

Here is an incomplete program (a program with some “holes”):

```
let rec foo (xs : int list) (n : int) : int =
  let y = ?1
  match xs with
  | []                -> ?2
  | x::xs' when ?3 -> ?4
  | _                 -> ?5
```

What are the holes and what can we fill them with?

Hole	Type	In scope
`?1`	any type	`xs, n, foo, ...`
`?2`	`int`	`xs, n, foo, y, ...`
`?3`	`bool`	`xs, n, foo, y, x, xs', ...`
`?4`	`int`	`xs, n, foo, y, x, xs', ...`
`?5`	`int`	`xs, n, foo, y, ...`

Part I: Discriminated Unions (motivation)

Suppose we wish to represent contact information.

An address can be:

- ```
  type PhysAddr = string * string * string
  let a1 : PhysAddr = "Ehitajate tee", "5", "Tallinn"
  ```
- ```
  type VirtAddr = string * string
  let a2 : VirtAddr = "juulius", "ttu.ee"
  ```

We wish to have a list of all addresses, both physical and virtual, of a contact.

Note that `a1 :: a2 :: []` does not typecheck. (Why?)

Goal: a type in which we can combine the two address types.

Informally, we would like to have something like

```
type Address = PhysAddr + VirtAddr
```

where the $+$ operator on types would be similar to disjoint union of sets.

In other words, `a : Address` could either be a physical or a virtual address but not both.

Part I: Discriminated Unions (defining, constructing)

Defining a type with distinct cases (this is the discriminating part):

```
type type-name =
  | case-identifier-1 of type-1-1 [ * type-1-2 ...]]
  ...
  | case-identifier-n of type-n-1 [ * type-n-2 ...]]
```

A simple example that combines the types `string`, `int` and `int`:

```
type StringsOrInts =
  | OfString of string
  | OfInt1   of int
  | OfInt2   of int
```

Note that:

- `OfString "abc" : StringsOrInts`
- The expression `OfString 17` does not typecheck
- `OfInt1 17 : StringsOrInts`
- `OfInt2 17 : StringsOrInts`
- `OfInt1 17 <> OfInt2 17`

Part I: Discriminated Unions (deconstructing)

In general, to use a value of a discriminated union type, we need to know which of the constructors (cases) was used to construct it and we have to say what to do in each case.

Suppose we are given an `soi : StringsOrInts` and we wish to compute from it a result of type `R`. To accomplish this we essentially have to describe three functions:

```
string -> R      // when soi = OfString s
int    -> R      // when soi = OfInt1 n
int    -> R      // when soi = OfInt2 n
```

We can use pattern matching for this.

Importantly, the `case-identifiers` can be used as patterns and we can also bind the values passed to the constructors.

A `case-indentifier` only matches itself.

```
let foo (soi : StringsOrInts) : bool =
  match soi with
   | OfString s            -> false
   | OfInt1   n when n < 0 -> true
   | OfInt1   n            -> n % 2 = 0
   | OfInt2   _            -> true
```

Enumeration Types as Discriminated Unions

The following is a type definition enumerating three colours:

```
type Colour = Red of unit | Green of unit | Blue of unit
```

We can also declare a constructor without any argument types

As `()` is the only value of type `unit`, an equivalent definition is:

```
type Colour = Red | Green | Blue
```

Using an enumeration type:

```
let show (c : Colour) : string =
  match c with
  | Red   -> "red"
  | Green -> "green"
  | Blue  -> "blue"
```

Discriminated Unions with a Single Constructor

The following is a valid type definition:

```
type DifferentInt = DifferentInt of int
```

This may seem rather useless: values of type `DifferentInt` are just values of type `int` labelled with the `DifferentInt` label, i.e., equivalent to `int`

```
let di2int (di : DifferentInt) : int =
  match di with
  | DifferentInt n -> n

di2int (DifferentInt 17) = 17
```

Such types can be useful:

```
type Meter    = M  of float
type Kilogram = Kg of float
type Second   = S  of float
```

We have three *different* types that are equivalent to type `float`. See also: units of measure.

The expression `M 1.0 : Second` does not typecheck

Recursive Discriminated Unions

Discriminated unions can be recursive: in the argument types to the constructor we can refer to the type we are defining

```
type IntList = INil | ICons of int * IntList
```

Note the `IntList` argument to `ICons` and the lack of arguments to `INil`.

```
let il1 : IntList = INil
let il2 : IntList = ICons (1, INil)
let il3 : IntList = ICons (1, ICons (2, ICons (3, INil)))
```

To use a value of a recursive DU we may need to use recursion:

```
let rec sum (il : IntList) : int =
  match il with
  | INil          -> 0
  | ICons (n, il') -> n + sum il'
```

Parameterised Discriminated Unions

Here is another recursive DU:

```
type StringList = SNil | SCons of string * StringList
```

This looks very similar to `IntList`

If we ignore the names we have chosen, then the only difference is the type of the first argument to the `Cons` constructor: `int` vs `string`

We can abstract out the type of this argument:

```
type 'a PList = Nil                        // []
              | Cons of 'a * 'a PList      // ::
```

This is essentially how lists are defined in F#.

We can think of `PList` as a function on types (something like `type -> type`): given a type `T` it gives us the type `T PList`.

Note that the type parameter can also be given in a different notation:

```
type PList<'a> = Nil                          // []
               | Cons of 'a * PList<'a>       // ::
```

The `option` types

Intuition: take an existing type and add one additional value to it.

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

The additional value (`None`) is often used to represent the lack of a value of type `'a`.

The `option` type is used quite extensively in the standard library.

You may already be familiar with somewhat similar ideas from other languages.

For example, in Java, the type (or class) `Integer` has as values (objects) all `int`s together with one additional value, `null`.

Java leaves the presence of this additional value implicit. Furthermore, it is not possible to have a reference type without this additional value.

With option types we can be precise about the absence/presence of such an additional value.

Another viewpoint: `'a option` can be seen as the type of lists of length at most one.

Examples

```
let noInt : int option = None
let anInt : int option = Some 11
```

A function that returns the index of the first occurrence of an element in a list:

```
let rec indexOf (el : 'a) (xs : 'a list) : int option =
  match xs with
  | []        -> None
  | x :: xs' -> if x = el
                then Some 0
                else match indexOf el xs' with
                     | None   -> None
                     | Some i -> Some (i + 1)
```

As the element might not be in the list, the result type is `int option`.

Note that we first match on the list `xs` to traverse the list.

We also match on the result of the recursive call `indexOf el xs'` to handle the cases when `el` was not in the tail `xs'` and when it was.

Records: aggregates of named values

```
type typename = { label_1 : type_1;
                  ...
                  label_n : type_n; }
```

Here is an example record type and two values of that type. When creating a record we must supply a value for each field.

```
type Person = { name : string; age : int }

let tõnu  = { name = "Tõnu"; age = 12 }
val tõnu : Person = { name = "Tõnu"; age = 12 }
tõnu.age // evaluates to 12

let tõnu2 = { Person.age = 12; name = "Tõnu" }
let bad   = { age = 12 }
```

We can create (copy) a modification of a record.

```
let mari = { tõnu with name = "Mari" }
val mari : Person = { name = "Mari"; age = 12 }
```

More records

Pattern matching on records.

```
let foo (p : Person) : string =
  match p with
  | { name = ""; age = _ } -> "empty string"
  | { name = n ; age = a } -> $"Name is {n} and age is {a}"
```

Records can be recursive and parameterized.

```
type 'a NotList = { head : 'a; tail : 'a NotList }

let rec x = { head = 1; tail = { head = 2; tail = x } }
x.head           // ?
x.tail.head      // ?
x.tail.tail.head // ?
```

This is not allowed! (But is allowed with `IntList`.)

```
let rec xs = 1 :: 2 :: xs
```

Part II: Lists

- Generation of lists
- Useful functions on lists
- Typical recursions on lists

Range expressions (1)

A simple range expression $[b \;..\; e]$, where $e \geq b$, generates the list:

$$[b; b+1; b+2; \ldots; b+n]$$

where n is chosen such that $b + n \leq e < b + n + 1$.

Example

```
[ -3 .. 5 ];;
val it : int list = [-3; -2; -1; 0; 1; 2; 3; 4; 5]
```

```
[2.4 .. 3.0 ** 1.7];;
val it : float list = [2.4; 3.4; 4.4; 5.4; 6.4]
```

Note that `3.0 ** 1.7 = 6.47300784`.

The range expression generates the empty list when $e < b$:

```
[7 .. 4];;
val it : int list = []
```

Range expressions (1)

A simple range expression $[b\ ..\ e]$, where $e \geq b$, generates the list:

$$[b; b+1; b+2; \ldots; b+n]$$

where n is chosen such that $b+n \leq e < b+n+1$.

Example

```
[ -3 .. 5 ];;
val it : int list = [-3; -2; -1; 0; 1; 2; 3; 4; 5]
```

```
[2.4 .. 3.0 ** 1.7];;
val it : float list = [2.4; 3.4; 4.4; 5.4; 6.4]
```

Note that `3.0 ** 1.7 = 6.47300784`.

The range expression generates the empty list when $e < b$:

```
[7 .. 4];;
val it : int list = []
```

Range expressions (1)

A simple range expression $[b\ ..\ e]$, where $e \geq b$, generates the list:

$$[b; b+1; b+2; \ldots; b+n]$$

where n is chosen such that $b+n \leq e < b+n+1$.

Example

```
[ -3 .. 5 ];;
val it : int list = [-3; -2; -1; 0; 1; 2; 3; 4; 5]
```

```
[2.4 .. 3.0 ** 1.7];;
val it : float list = [2.4; 3.4; 4.4; 5.4; 6.4]
```

Note that `3.0 ** 1.7 = 6.47300784`.

The range expression generates the empty list when $e < b$:

```
[7 .. 4];;
val it : int list = []
```

Range expressions (1)

A simple range expression $[b \;..\; e]$, where $e \geq b$, generates the list:

$$[b; b+1; b+2; \ldots; b+n]$$

where n is chosen such that $b+n \leq e < b+n+1$.

Example

```
[ -3 .. 5 ];;
val it : int list = [-3; -2; -1; 0; 1; 2; 3; 4; 5]
```

```
[2.4 .. 3.0 ** 1.7];;
val it : float list = [2.4; 3.4; 4.4; 5.4; 6.4]
```

Note that `3.0 ** 1.7 = 6.47300784`.

The range expression generates the empty list when $e < b$:

```
[7 .. 4];;
val it : int list = []
```

Range expressions (1)

A simple range expression $[b \, .. \, e]$, where $e \geq b$, generates the list:

$$[b; b+1; b+2; \ldots; b+n]$$

where n is chosen such that $b + n \leq e < b + n + 1$.

Example

```
[ -3 .. 5 ];;
val it : int list = [-3; -2; -1; 0; 1; 2; 3; 4; 5]


[2.4 .. 3.0 ** 1.7];;
val it : float list = [2.4; 3.4; 4.4; 5.4; 6.4]
```

Note that `3.0 ** 1.7 = 6.47300784`.

The range expression generates the empty list when $e < b$:

```
[7 .. 4];;
val it : int list = []
```

Range expressions (2)

The range expression $[b\ ..\ s\ ..\ e]$ generates either an ascending or a descending list:

$$[b\ ..\ s\ ..\ e] \;=\; [b; b+s; b+2s; \ldots; b+ns]$$

$$\text{where} \qquad \begin{cases} b+ns \le e < b+(n+1)s & \text{if } s > 0 \\ b+ns \ge e > b+(n+1)s & \text{if } s < 0 \end{cases}$$

depending on the sign of s.

Examples:

```
[6 .. -1 .. 2];;
val it : int list = [6; 5; 4; 3; 2]
```

and the float representation of $0, \pi/2, \pi, \frac{3}{2}\pi, 2\pi$ is generated by:

```
[0.0 .. System.Math.PI/2.0 .. 2.0*System.Math.PI];;
val it : float list =
  [0.0; 1.570796327; 3.141592654; 4.71238898; 6.283185307]
```

Range expressions (2)

The range expression $[b \,..\, s \,..\, e]$ generates either an ascending or a descending list:

$$[b \,..\, s \,..\, e] \; = \; [b; b+s; b+2s; \ldots; b+ns]$$

$$\text{where} \qquad \begin{cases} b+ns \leq e < b+(n+1)s & \text{if } s>0 \\ b+ns \geq e > b+(n+1)s & \text{if } s<0 \end{cases}$$

depending on the sign of s.

Examples:

```
[6 .. -1 .. 2];;
val it : int list = [6; 5; 4; 3; 2]
```

and the float representation of $0, \pi/2, \pi, \frac{3}{2}\pi, 2\pi$ is generated by:

```
[0.0 .. System.Math.PI/2.0 .. 2.0*System.Math.PI];;
val it : float list =
  [0.0; 1.570796327; 3.141592654; 4.71238898; 6.283185307]
```

Range expressions (2)

The range expression $[b .. s .. e]$ generates either an ascending or a descending list:

$$[b .. s .. e] \;=\; [b; b+s; b+2s; \ldots; b+ns]$$

$$\text{where} \qquad \begin{cases} b+ns \le e < b+(n+1)s & \text{if } s > 0 \\ b+ns \ge e > b+(n+1)s & \text{if } s < 0 \end{cases}$$

depending on the sign of s.

Examples:

```
[6 .. -1 .. 2];;
val it : int list = [6; 5; 4; 3; 2]
```

and the float representation of $0, \pi/2, \pi, \frac{3}{2}\pi, 2\pi$ is generated by:

```
[0.0 .. System.Math.PI/2.0 .. 2.0*System.Math.PI];;
val it : float list =
  [0.0; 1.570796327; 3.141592654; 4.71238898; 6.283185307]
```

Simple recursion on lists

We consider now three simple functions:

- append
- reverse
- isMember

whose declarations follow the structure of list argument

```
let rec f ... xs ... =
    | []     -> v
    | x::xs -> .... f xs ...
```

using just two clauses.

Append

The infix operator `@` (called 'append') joins two lists:

$$\begin{aligned}[x_1; x_2; \ldots; x_m] \ @ \ [y_1; y_2; \ldots; y_n] \\ = [x_1; x_2; \ldots; x_m; y_1; y_2; \ldots; y_n]\end{aligned}$$

Properties

$$\begin{aligned}[\,] \ @ \ ys &= ys \\ [x_1; x_2; \ldots; x_m] \ @ \ ys &= x_1 :: ([x_2; \ldots; x_m] \ @ \ ys)\end{aligned}$$

Declaration

```
let rec (@) xs ys =
  match xs with
  | []       -> ys
  | x::xs' -> x::(xs' @ ys);;
val (@) : 'a list -> 'a list -> 'a list
```

Append

The infix operator `@` (called 'append') joins two lists:

$$\begin{aligned}[x_1; x_2; \ldots; x_m] \mathbin{@} [y_1; y_2; \ldots; y_n] \\ = [x_1; x_2; \ldots; x_m; y_1; y_2; \ldots; y_n]\end{aligned}$$

Properties

$$\begin{aligned}[\,] \mathbin{@} ys &= ys \\ [x_1; x_2; \ldots; x_m] \mathbin{@} ys &= x_1 :: ([x_2; \ldots; x_m] \mathbin{@} ys)\end{aligned}$$

Declaration

```
let rec (@) xs ys =
  match xs with
  | []      -> ys
  | x::xs' -> x::(xs' @ ys);;
val (@) : 'a list -> 'a list -> 'a list
```

Append

The infix operator `@` (called 'append') joins two lists:

$$[x_1; x_2; \ldots; x_m] \mathbin{@} [y_1; y_2; \ldots; y_n] \\ = [x_1; x_2; \ldots; x_m; y_1; y_2; \ldots; y_n]$$

Properties

$$\begin{aligned} [\,] \mathbin{@} ys &= ys \\ [x_1; x_2; \ldots; x_m] \mathbin{@} ys &= x_1 :: ([x_2; \ldots; x_m] \mathbin{@} ys) \end{aligned}$$

Declaration

```
let rec (@) xs ys =
  match xs with
  | []      -> ys
  | x::xs' -> x::(xs' @ ys);;
val (@) : 'a list -> 'a list -> 'a list
```

Append: evaluation

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

Evaluation

$$
\begin{array}{lll}
 & \texttt{[1;2] @ [3;4]} & \\
\rightsquigarrow & \texttt{1::([2] @ [3,4])} & (\mathtt{x} \mapsto 1,\ \mathtt{xs}' \mapsto [2],\ \mathtt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::(2::([] @ [3;4]))} & (\mathtt{x} \mapsto 2,\ \mathtt{xs}' \mapsto [\,],\ \mathtt{ys} \mapsto [3;4]) \\
\rightsquigarrow & \texttt{1::(2::[3;4])} & (\mathtt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::[2;3;4]} & \\
\rightsquigarrow & \texttt{[1;2;3;4]} &
\end{array}
$$

- Execution time is linear in the size of the first list
- Be careful with this usage: `xs @ [x]`

Append: evaluation

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

Evaluation

$$
\begin{array}{lll}
 & \texttt{[1;2] @ [3;4]} & \\
\rightsquigarrow & \texttt{1::([2] @ [3,4])} & (\mathtt{x} \mapsto 1,\ \mathtt{xs}' \mapsto [2],\ \mathtt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::(2::([] @ [3;4]))} & (\mathtt{x} \mapsto 2,\ \mathtt{xs}' \mapsto [\,],\ \mathtt{ys} \mapsto [3;4]) \\
\rightsquigarrow & \texttt{1::(2::[3;4])} & (\mathtt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::[2;3;4]} & \\
\rightsquigarrow & \texttt{[1;2;3;4]} &
\end{array}
$$

- Execution time is linear in the size of the first list
- Be careful with this usage: `xs @ [x]`

Append: polymorphic type

The answer from the system is:

```
> val (@) : 'a list -> 'a list -> 'a list
```

- `'a` is a *type variable*
- The type of `@` is *polymorphic* — it has many forms

`'a = int`: Appending integer lists

```
[1;2] @ [3;4];;
val it : int list = [1;2;3;4]
```

`'a = int list`: Appending lists of integer list

```
[[1];[2;3]] @ [[4]];;
val it : int list list = [[1]; [2; 3]; [4]]
```

`@` is a built-in function

Append: polymorphic type

The answer from the system is:

```
> val (@) : 'a list -> 'a list -> 'a list
```

- `'a` is a *type variable*
- The type of `@` is *polymorphic* — it has many forms

`'a = int`: Appending integer lists

```
[1;2] @ [3;4];;
val it : int list = [1;2;3;4]
```

`'a = int list`: Appending lists of integer list

```
[[1];[2;3]] @ [[4]];;
val it : int list list = [[1]; [2; 3]; [4]]
```

`@` is a built-in function

Append: polymorphic type

The answer from the system is:

```
> val (@) : 'a list -> 'a list -> 'a list
```

- `'a` is a *type variable*
- The type of `@` is *polymorphic* — it has many forms

`'a = int`: Appending integer lists

```
[1;2] @ [3;4];;
val it : int list = [1;2;3;4]
```

`'a = int list`: Appending lists of integer list

```
[[1];[2;3]] @ [[4]];;
val it : int list list = [[1]; [2; 3]; [4]]
```

`@` is a built-in function

Reverse

```
rev [x1; x2; ... ; xn] = [xn; ... ; x2; x1]
```

$\texttt{rev}\ [x_1; x_2; \ldots; x_n] = [x_n; \ldots; x_2; x_1]$

```
let rec naive_rev lst = match lst with
  | []    -> []
  | x::xs -> naive_rev xs @ [x];;
val naive_rev : 'a list -> 'a list
```

An evaluation:

```
   naive_rev[1;2;3]
⇝ naive_rev[2;3] @ [1]
⇝ (naive_rev[3] @ [2]) @ [1]
⇝ ((naive_rev[] @ [3]) @ [2]) @ [1]
⇝ (([] @ [3]) @ [2]) @ [1]
⇝ ([3] @ [2]) @ [1]
⇝ (3::([] @ [2])) @ [1]
⇝ (3::[2]) @ [1]
⇝ [3;2] @ [1]
⇝ 3::([2] @ [1])
⇝ ...
⇝ [3;2;1]
```

Takes $O(n^2)$ time — Built-in version (`List.rev`) is efficient $O(n)$

We consider efficiency later.

Reverse $\texttt{rev}\ [x_1; x_2; \ldots; x_n] = [x_n; \ldots; x_2; x_1]$

```
let rec naive_rev lst = match lst with
  | []    -> []
  | x::xs -> naive_rev xs @ [x];;
val naive_rev : 'a list -> 'a list
```

An evaluation:

```
    naive_rev[1;2;3]
⇝  naive_rev[2;3] @ [1]
⇝  (naive_rev[3] @ [2]) @ [1]
⇝  ((naive_rev[] @ [3]) @ [2]) @ [1]
⇝  (([] @ [3]) @ [2]) @ [1]
⇝  ([3] @ [2]) @ [1]
⇝  (3::([] @ [2])) @ [1]
⇝  (3::[2]) @ [1]
⇝  [3;2] @ [1]
⇝  3::([2] @ [1])
⇝  ...
⇝  [3;2;1]
```

Takes $O(n^2)$ time — Built-in version (`List.rev`) is efficient $O(n)$

We consider efficiency later.

Reverse

$\texttt{rev}\ [x_1; x_2; \ldots; x_n] = [x_n; \ldots; x_2; x_1]$

```
let rec naive_rev lst = match lst with
  | []     -> []
  | x::xs -> naive_rev xs @ [x];;
val naive_rev : 'a list -> 'a list
```

An evaluation:

```
    naive_rev[1;2;3]
⇝   naive_rev[2;3] @ [1]
⇝   (naive_rev[3] @ [2]) @ [1]
⇝   ((naive_rev[] @ [3]) @ [2]) @ [1]
⇝   (([] @ [3]) @ [2]) @ [1]
⇝   ([3] @ [2]) @ [1]
⇝   (3::([] @ [2])) @ [1]
⇝   (3::[2]) @ [1]
⇝   [3;2] @ [1]
⇝   3::([2] @ [1])
⇝   ...
⇝   [3;2;1]
```

Takes $O(n^2)$ time — Built-in version (`List.rev`) is efficient $O(n)$
We consider efficiency later.

Reverse

$\texttt{rev}\ [x_1; x_2; \ldots; x_n] = [x_n; \ldots; x_2; x_1]$

```
let rec naive_rev lst = match lst with
 | []    -> []
 | x::xs -> naive_rev xs @ [x];;
val naive_rev : 'a list -> 'a list
```

An evaluation:

```
   naive_rev[1;2;3]
⇝  naive_rev[2;3] @ [1]
⇝  (naive_rev[3] @ [2]) @ [1]
⇝  ((naive_rev[] @ [3]) @ [2]) @ [1]
⇝  (([] @ [3]) @ [2]) @ [1]
⇝  ([3] @ [2]) @ [1]
⇝  (3::([] @ [2])) @ [1]
⇝  (3::[2]) @ [1]
⇝  [3;2] @ [1]
⇝  3::([2] @ [1])
⇝  ...
⇝  [3;2;1]
```

Takes $O(n^2)$ time — Built-in version (`List.rev`) is efficient $O(n)$
We consider efficiency later.

Membership — equality constraint

$$\begin{aligned} & \texttt{isMember}\ x\ [y_1; y_2; \ldots; y_n] \\ = \quad & (x = y_1) \lor (x = y_2) \lor \cdots \lor (x = y_n) \\ = \quad & (x = y_1) \lor (\texttt{isMember}\ x\ [y_2; \ldots; y_n]) \end{aligned}$$

Declaration

```
let rec isMember x lst = match lst with
| []     -> false
| y::ys -> x=y || isMember x ys;;
val isMember : 'a -> 'a list -> bool when  'a : equality
```

- `'a` is a type variable
- `'a` must satisfy the constraint `equality`, i.e., it must be possible to compare values of that type for equality
- Function types do not satisfy `equality`

`isMember (1,true) [(2,true); (1,false)]` ⇝ false
`isMember [1;2;3] [[1]; []; [1;2;3]]` ⇝ true

Membership — equality constraint

$$
\begin{aligned}
& \texttt{isMember}\ x\ [y_1; y_2; \ldots; y_n] \\
= \ & (x = y_1) \lor (x = y_2) \lor \cdots \lor (x = y_n) \\
= \ & (x = y_1) \lor (\texttt{isMember}\ x\ [y_2; \ldots; y_n])
\end{aligned}
$$

Declaration

```
let rec isMember x lst = match lst with
| []     -> false
| y::ys -> x=y || isMember x ys;;
val isMember : 'a -> 'a list -> bool when  'a : equality
```

- `'a` is a type variable
- `'a` must satisfy the constraint `equality`, i.e., it must be possible to compare values of that type for equality
- Function types do not satisfy `equality`

`isMember (1,true) [(2,true); (1,false)]` ⇝ false
`isMember [1;2;3] [[1]; []; [1;2;3]]` ⇝ true

Membership — equality constraint

$$
\begin{aligned}
& \texttt{isMember}\ x\ [y_1; y_2; \ldots; y_n] \\
=\ & (x = y_1) \lor (x = y_2) \lor \cdots \lor (x = y_n) \\
=\ & (x = y_1) \lor (\texttt{isMember}\ x\ [y_2; \ldots; y_n])
\end{aligned}
$$

Declaration

```
let rec isMember x lst = match lst with
| []     -> false
| y::ys -> x=y || isMember x ys;;
val isMember : 'a -> 'a list -> bool when  'a : equality
```

- `'a` is a type variable
- `'a` must satisfy the constraint `equality`, i.e., it must be possible to compare values of that type for equality
- Function types do not satisfy `equality`

`isMember (1,true) [(2,true); (1,false)]` $\rightsquigarrow$ false
`isMember [1;2;3] [[1]; []; [1;2;3]]` $\rightsquigarrow$ true

Match on results of recursive call

We consider declarations on the form:

let rec f ... xs ... =
    ....
    let $pat(\overline{y})$ = f xs
    $e(\overline{y})$

Recall unzip from last week.

Example: `sumProd`

$$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}] = (x_0 + x_1 + \ldots + x_{n-1}\ ,\ x_0 * x_1 * \ldots * x_{n-1})$$

The declaration is based on the recursion formula:

$$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}] = (x_0 + \texttt{rSum}, x_0 * \texttt{rProd})$$
where $(\texttt{rSum}, \texttt{rProd}) = \texttt{sumProd}\ [x_1; \ldots; x_{n-1}]$

This gives the declaration

```
let rec sumProd lst = match lst with
    | []       -> (0,1)
    | x::rest ->
          let (rSum,rProd) = sumProd rest
          (x+rSum,x*rProd);;
val sumProd : int list -> int * int

sumProd [2;5];;
val it : int * int = (7, 10)
```

Example: sumProd

$$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}] = (\ x_0 + x_1 + \ldots + x_{n-1}\ ,\ x_0 * x_1 * \ldots * x_{n-1}\ )$$

The declaration is based on the recursion formula:

$$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}] = (x_0 + \texttt{rSum}, x_0 * \texttt{rProd})$$
$$\text{where } (\texttt{rSum}, \texttt{rProd}) = \texttt{sumProd}\ [x_1; \ldots; x_{n-1}]$$

This gives the declaration

```
let rec sumProd lst = match lst with
    | []        -> (0,1)
    | x::rest ->
            let (rSum,rProd) = sumProd rest
            (x+rSum,x*rProd);;
val sumProd : int list -> int * int

sumProd [2;5];;
val it : int * int = (7, 10)
```

Example: `sumProd`

$$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}] = (\ x_0 + x_1 + \ldots + x_{n-1}\ ,\ x_0 * x_1 * \ldots * x_{n-1}\ )$$

The declaration is based on the recursion formula:

$$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}] = (x_0 + \texttt{rSum}, x_0 * \texttt{rProd})$$
$$\text{where } (\texttt{rSum}, \texttt{rProd}) = \texttt{sumProd}\ [x_1; \ldots; x_{n-1}]$$

This gives the declaration

```
let rec sumProd lst = match lst with
    | []        -> (0,1)
    | x::rest ->
            let (rSum,rProd) = sumProd rest
            (x+rSum,x*rProd);;
val sumProd : int list -> int * int

sumProd [2;5];;
val it : int * int = (7, 10)
```

Example: `split`

Declare a function `split` such that:

$$\texttt{split}\ [x_0; x_1; x_2; x_3; \ldots; x_{n-1}] = ([x_0; x_2; \ldots], [x_1; x_3; \ldots])$$

The declaration is

```
let rec split lst = match lst with
    | []         -> ([],[])
    | [x]        -> ([x],[])
    | x::y::xs -> let (xs1,xs2) = split xs
                  in (x::xs1,y::xs2);;
```

Notice

- a convenient division into three cases, and
- the recursion formula

$$\texttt{split}\ [x_0; x_1; x_2; \ldots; x_{n-1}] = (x_0 :: \texttt{xs1}, x_1 :: \texttt{xs2})$$
where $(\texttt{xs1},\texttt{xs2}) = \texttt{split}\ [x_2; \ldots; x_{n-1}]$

Example: `split`

Declare a function `split` such that:

$$\texttt{split}\ [x_0; x_1; x_2; x_3; \ldots; x_{n-1}] = ([x_0; x_2; \ldots], [x_1; x_3; \ldots])$$

The declaration is

```
let rec split lst = match lst with
    | []        -> ([],[])
    | [x]       -> ([x],[])
    | x::y::xs -> let (xs1,xs2) = split xs
                  in (x::xs1,y::xs2);;
```

Notice

- a convenient division into three cases, and
- the recursion formula

$$\texttt{split}\ [x_0; x_1; x_2; \ldots; x_{n-1}] = (x_0 :: \texttt{xs1}, x_1 :: \texttt{xs2})$$
$$\text{where } (\texttt{xs1}, \texttt{xs2}) = \texttt{split}\ [x_2; \ldots; x_{n-1}]$$

Example: split

Declare a function `split` such that:

$$\texttt{split [}x_0\texttt{;}x_1\texttt{;}x_2\texttt{;}x_3\texttt{; ... ;}x_{n-1}\texttt{]} = (\texttt{[}x_0\texttt{;}x_2\texttt{; ...]}, \texttt{[}x_1\texttt{;}x_3\texttt{; ...]})$$

The declaration is

```
let rec split lst = match lst with
    | []         -> ([],[])
    | [x]        -> ([x],[])
    | x::y::xs -> let (xs1,xs2) = split xs
                  in (x::xs1,y::xs2);;
```

Notice

- a convenient division into three cases, and
- the recursion formula

$$\texttt{split [}x_0\texttt{;}x_1\texttt{;}x_2\texttt{; ... ;}x_{n-1}\texttt{]} = (x_0 \texttt{ :: xs1}, x_1 \texttt{ :: xs2})$$
$$\text{where } \texttt{(xs1,xs2)} = \texttt{split [}x_2\texttt{; ... ;}x_{n-1}\texttt{]}$$

Part III: Higher-order list functions

Higher-order functions: functions that accept a function as an argument or return a function as a result

Higher-order functions are:

- everywhere

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

- powerful

  Parameterized modules, succinct code, ...

HIGHER-ORDER FUNCTIONS ARE USEFUL

Part III: Higher-order list functions

Higher-order functions: functions that accept a function as an argument or return a function as a result

Higher-order functions are:

- everywhere

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

- powerful

  Parameterized modules, succinct code, ...

HIGHER-ORDER FUNCTIONS ARE USEFUL

Part III: Higher-order list functions

Higher-order functions: functions that accept a function as an argument or return a function as a result

Higher-order functions are:

- everywhere

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

- powerful

  Parameterized modules, succinct code, ...

HIGHER-ORDER FUNCTIONS ARE USEFUL

Now down to earth

Many recursive declarations follows the same schema, for example:

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

Goal:	avoid repeating (almost) identical code fragments (i.e., patterns)
Solution:	capture frequently occurring patterns as higher-order functions
Result:	more succinct declarations using higher-order functions

Today we take a look at higher-order list functions:

- `map`
- `filter`
- `fold`, `foldBack`
- `exists`, `forall`, `tryFind`

Now down to earth

Many recursive declarations follows the same schema, for example:

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

Goal: avoid repeating (almost) identical code fragments (i.e., patterns)

Solution: capture frequently occurring patterns as higher-order functions

Result: more succinct declarations using higher-order functions

Today we take a look at higher-order list functions:

- `map`
- `filter`
- `fold`, `foldBack`
- `exists`, `forall`, `tryFind`

Now down to earth

Many recursive declarations follows the same schema, for example:

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

Goal: avoid repeating (almost) identical code fragments (i.e., patterns)

Solution: capture frequently occurring patterns as higher-order functions

Result: more succinct declarations using higher-order functions

Today we take a look at higher-order list functions:

- `map`
- `filter`
- `fold`, `foldBack`
- `exists`, `forall`, `tryFind`

Two simple list functions

```
let rec add n xs =
  match xs with
  | []        -> []
  | x :: xs' -> n + x :: add n xs'

let rec flip xs =
  match xs with
  | []        -> []
  | x :: xs' -> -1 * x :: flip xs'
```

```
add: int -> int list -> int list
flip:       int list -> int list
```

```
add 5 [-2; -1; 0; 1; 2] = [3; 4; 5; 6; 7]
flip  [-2; -1; 0; 1; 2] = [2; 1; 0; -1; -2]
```

The only difference is what happens to the elements of the list: `n + x` vs `-1 * x`.

What happens to the elements can be described by a function of type `int -> int`:
`fun x -> n + x` vs `fun x -> -1 * x`.

The `map` pattern

```
let rec mapInts (f : int -> int) (xs : int list) : int list =
  match xs with
  | []        -> []
  | x :: xs' -> f x :: mapInts f xs'
```

We are not doing anything specific to `int` in this definition. Let's remove the type annotations and see what type is inferred from the definition.

```
let rec map f xs =
  match xs with
  | []        -> []
  | x :: xs' -> f x :: map f xs'
```

```
val map: f: ('a -> 'b) -> xs: 'a list -> 'b list
```

We can now say the following (without using `rec`).

```
let add n xs = map (fun x ->  n + x) xs
let flip  xs = map (fun x -> -1 * x) xs
```

The `map` pattern

```
let rec mapInts (f : int -> int) (xs : int list) : int list =
  match xs with
  | []        -> []
  | x :: xs' -> f x :: mapInts f xs'
```

We are not doing anything specific to `int` in this definition. Let's remove the type annotations and see what type is inferred from the definition.

```
let rec map f xs =
  match xs with
  | []        -> []
  | x :: xs' -> f x :: map f xs'
```

```
val map: f: ('a -> 'b) -> xs: 'a list -> 'b list
```

We can now say the following (without using `rec`).

```
let add n xs = map (fun x ->  n + x) xs
let flip  xs = map (fun x -> -1 * x) xs
```

The `map` pattern

```
let rec mapInts (f : int -> int) (xs : int list) : int list =
  match xs with
  | []        -> []
  | x :: xs' -> f x :: mapInts f xs'
```

We are not doing anything specific to `int` in this definition. Let's remove the type annotations and see what type is inferred from the definition.

```
let rec map f xs =
  match xs with
  | []        -> []
  | x :: xs' -> f x :: map f xs'

val map: f: ('a -> 'b) -> xs: 'a list -> 'b list
```

We can now say the following (without using `rec`).

```
let add n xs = map (fun x ->  n + x) xs
let flip  xs = map (fun x -> -1 * x) xs
```

More examples

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

let isPositive xs  = map (fun x      -> x > 0) xs
let addFstSnd  xys = map (fun (x, y) -> x + y) xys

isPositive :           int list -> bool list
addFstSnd  : (int * int) list -> int  list

isPositive [-1; 0; 1]                  = [false; false; true]
addFstSnd  [(-1, 2); (1, 3); (2, 4)] = [1; 4; 6]
```

The `map` pattern applies the same function (transformation) to every element in the input list.

$$\texttt{map}\ f\ [v_1; \ldots; v_n] \ = \ [f\ v_1; \ldots; f\ v_n]$$

Another viewpoint: from a function of type `'a -> 'b` construct a function of type `'a list -> 'b list`.

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

Exercise

Declare a function

$$\texttt{g}\ [x_1; \ldots; x_n] = [x_1^2 + 1; \ldots; x_n^2 + 1]$$

Remember

$$\texttt{map}\ f\ [v_1; \ldots; v_n] \ = \ [f\ v_1; \ldots; f\ v_n]$$

There is also a `map` operation for `option` types. What does it do?

```
Option.map : ('a -> 'b) -> 'a option -> 'b option
```

Something similar to `map`

Recall the viewpoint that `map` constructs a function of type `'a list -> 'b list` from a function of type `'a -> 'b`.

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

In the `List` module there is the following function.

```
List.collect : ('a -> 'b list) -> ('a list -> 'b list)
```

What does it do? How to construct a function of type `'a list -> 'b list` given a function of type `'a -> 'b list`?

For example, what is (or should be) the result of evaluating the following?

```
List.collect (fun x -> [x; x - 1]) [1 .. 5]
```

Higher-order list functions: `filter`

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

`filter p xs` is the sublist of those elements `x` of `xs` for which `p x = true`

Declaration **Library function**

```
let rec filter p xs =
  match xs with
  | []        -> []
  | 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 ['2'; 'p'; 'F'; '-']
val it : char list = ['p'; 'F']
```

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

Higher-order list functions: `filter`

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

`filter p xs` is the sublist of those elements `x` of `xs` for which `p x = true`

Declaration

Library function

```
let rec filter p xs =
  match xs with
  | []        -> []
  | 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 ['2'; 'p'; 'F'; '-']
val it : char list = ['p'; 'F']
```

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

Higher-order list functions: `filter`

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

`filter p xs` is the sublist of those elements `x` of `xs` for which `p x = true`

Declaration **Library function**

```
let rec filter p xs =
  match xs with
  | []         -> []
  | 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 ['2'; 'p'; 'F'; '-']
val it : char list = ['p'; 'F']
```

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

Higher-order list functions: `filter`

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

`filter p xs` is the sublist of those elements `x` of `xs` for which `p x = true`

Declaration

Library function

```
let rec filter p xs =
  match xs with
  | []         -> []
  | 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 ['2'; 'p'; 'F'; '-']
val it : char list = ['p'; 'F']
```

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

Combining a list of values to a single value

Add the elements in a list of integers:

```
let rec sum (xs : int list) : int =
  match xs with
  | []        -> 0
  | x :: xs' -> x + sum xs'
```

Multiply the absolute values of a list of integers:

```
let rec prodabs (xs : int list) : int =
  match xs with
  | []        -> 1
  | x :: xs' -> abs x * prodabs xs'
```

Rather small differences:

- the base value: `0` vs `1`
- the function applied to `x`: `id` vs `abs`
- the function used to combine with result of recursive call: `+` vs `*`

A first generalisation

Capture the pattern in `sum` and `prodabs` by abstracting out the differences:

- the base value `b`
- the function `f` combining an element `x` and a partial result `r`

```
let rec combineInts (f  : int -> int -> int)
                    (xs : int list)
                    (b  : int)
                    : int =
  match xs with
  | []        -> b
  | x :: xs' -> f x (combineInts f xs' b)

let sum     xs = combineInts (fun x r -> x     + r) xs 0
let prodabs xs = combineInts (fun x r -> abs x * r) xs 1
```

Observe that we do not do anything that is specific to integers in the definition of `combineInts`. Thus we have no reason to restrict this pattern to integers.

Folding a list (from the right)

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

let sum     xs = foldBack (fun x r -> x     + r) xs 0
let prodabs xs = foldBack (fun x r -> abs x * r) xs 1
```

We fold *back* (or from the right) in the sense that the base value “goes” to the right of the list and we start combining from there.

```
sum [1; 2; 3] =  foldBack (+) [1; 2; 3] 0
              ~> 1 + (foldBack (+) [2; 3] 0)
              ~> 1 + (2 + (foldBack (+) [3] 0))
              ~> 1 + (2 + (3 + (foldBack (+) [] 0)))
              ~> 1 + (2 + (3 + 0))
              ~> 6
```

Higher-order list functions: `foldBack`

Suppose that $\otimes$ is some infix function. Then `foldBack` operates as:

$$\texttt{foldBack}\ (\otimes)\ [a_0;\ a_1;\ \ldots\ ;\ a_{n-2};\ a_{n-1}]\ e_b$$
$$= \quad a_0 \otimes (a_1 \otimes (\ldots (a_{n-2} \otimes (a_{n-1} \otimes e_b)) \ldots))$$

Many functions can be expressed as a `foldBack`:

- `let conj xs = foldBack (&&) xs true`
- `let (@) xs ys = foldBack (fun x r -> x :: r) xs ys`
- `let length xs = foldBack (fun _ r -> 1 + r) xs 0`
- `let map f xs = foldBack (fun x r -> f x :: r) xs []`

Evaluating a `map` that is defined via `foldBack`:

```
map abs [-1; 2; -3]
  =  foldBack fn [-1; 2; -3] []
  ~> abs -1 :: foldBack fn [2; -3] []
  ~> abs -1 :: abs 2 :: foldBack fn [-3] []
  ~> abs -1 :: abs 2 :: abs -3 :: foldBack fn [] []
  ~> abs -1 :: abs 2 :: abs -3 :: []
  ~> 1      :: 2     :: 3      :: []
```

where `fn = fun x r -> abs x :: r`

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

We can also fold from the left:

```
let rec fold (f  : 'b -> 'a -> 'b)
             (r  : 'b)
             (xs : 'a list)
             : 'b =
  match xs with
  | []        -> r
  | x :: xs' -> fold f (f r x) xs'
```

Example: using cons in connection with `fold` gives the reverse function:

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

This function has linear execution time:

```
rev [1; 2; 3]
  =  fold fn []                      [1; 2 ;3]
  ~> fold fn (1 :: [])               [2; 3]
  ~> fold fn (2 :: 1 :: [])          [3]
  ~> fold fn (3 :: 2 :: 1 :: []) []
  ~> 3 :: 2 :: 1 :: []
```

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

We can also fold from the left:

```
let rec fold (f  : 'b -> 'a -> 'b)
             (r  : 'b)
             (xs : 'a list)
             : 'b =
  match xs with
  | []        -> r
  | x :: xs' -> fold f (f r x) xs'
```

Example: using cons in connection with `fold` gives the reverse function:

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

This function has linear execution time:

```
rev [1; 2; 3]
  =  fold fn []                      [1; 2 ;3]
  ~> fold fn (1 :: [])               [2; 3]
  ~> fold fn (2 :: 1 :: [])          [3]
  ~> fold fn (3 :: 2 :: 1 :: [])     []
  ~> 3 :: 2 :: 1 :: []
```

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

We can also fold from the left:

```
let rec fold (f  : 'b -> 'a -> 'b)
             (r  : 'b)
             (xs : 'a list)
             : 'b =
  match xs with
  | []        -> r
  | x :: xs' -> fold f (f r x) xs'
```

Example: using cons in connection with `fold` gives the reverse function:

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

This function has linear execution time:

```
rev [1; 2; 3]
  =  fold fn []                       [1; 2 ;3]
  ~> fold fn (1 :: [])                [2; 3]
  ~> fold fn (2 :: 1 :: [])           [3]
  ~> fold fn (3 :: 2 :: 1 :: []) []
  ~> 3 :: 2 :: 1 :: []
```

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

Suppose that $\oplus$ is an infix function.

Then the `fold` function operates as:

$$\begin{aligned} &\texttt{fold}\ (\oplus)\ e_b\ [a_0;\ a_1;\ \ldots\ ;\ a_{n-2};\ a_{n-1}\ ] \\ &= ((\ldots((e_b \oplus a_0) \oplus a_1)\ldots) \oplus a_{n-2}) \oplus a_{n-1} \end{aligned}$$

In other words, it applies $\oplus$ from left to right.

Compare:

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

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


fold     (fun r x -> x :: r) [] [1;2;3;4;5]

foldBack (fun x r -> x :: r) [1;2;3;4;5] []
```

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

Suppose that $\oplus$ is an infix function.

Then the `fold` function operates as:

$$\begin{aligned} &\texttt{fold}\ (\oplus)\ e_b\ [a_0;\ a_1;\ \ldots\ ;\ a_{n-2};\ a_{n-1}\ ] \\ &= ((\ldots((e_b \oplus a_0) \oplus a_1)\ldots) \oplus a_{n-2}) \oplus a_{n-1} \end{aligned}$$

In other words, it applies $\oplus$ from left to right.

Compare:

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

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


fold     (fun r x -> x :: r) [] [1;2;3;4;5]

foldBack (fun x r -> x :: r) [1;2;3;4;5] []
```

Higher-order list functions: `exists`

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

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

Declaration **Library function**

```
let rec exists p xs =
  match xs with
  | []        -> 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
```

Exercise: define `exists` as a fold. Compare it to this definition.

Higher-order list functions: `exists`

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

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

Declaration

Library function

```
let rec exists p xs =
  match xs with
  | []         -> 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
```

Exercise: define `exists` as a fold. Compare it to this definition.

Higher-order list functions: `exists`

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

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

Declaration

Library function

```
let rec exists p xs =
  match xs with
  | []         -> 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
```

Exercise: define `exists` as a fold. Compare it to this definition.

Higher-order list functions: `exists`

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

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

Declaration

Library function

```
let rec exists p xs =
  match xs with
  | []        -> 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
```

Exercise: define `exists` as a fold. Compare it to this definition.

Exercise

Declare `isMember` function using `exists`.

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

Remember

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

Higher-order list functions: `forall`

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

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

Declaration

Library function

```
let rec forall p xs =
  match xs with
  | []        -> 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
```

Higher-order list functions: `forall`

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

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

Declaration

Library function

```
let rec forall p xs =
  match xs with
  | []        -> 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
```

Higher-order list functions: `forall`

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

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

Declaration

Library function

```
let rec forall p xs =
  match xs with
  | []         -> 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
```

Higher-order list functions: `forall`

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

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

Declaration

Library function

```
let rec forall p xs =
  match xs with
  | []         -> 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
```

Exercises

Declare a function

$$\texttt{disjoint}\ xs\ ys$$

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

Declare a function

$$\texttt{subset}\ xs\ ys$$

which is true when every element in the lists xs is in ys, and false otherwise.

Remember

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

Exercises

Declare a function

$$\texttt{disjoint}\ xs\ ys$$

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

Declare a function

$$\texttt{subset}\ xs\ ys$$

which is true when every element in the lists *xs* is in *ys*, and false otherwise.

Remember

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

Exercise

Declare a function

$$\texttt{inter}\ xs\ ys$$

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

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

Higher-order list functions: `tryFind`

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

```
let rec tryFind p xs =
  match xs with
   | 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
```

Higher-order list functions: `tryFind`

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

```
let rec tryFind p xs =
  match xs with
  | 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
```

Higher-order list functions: `tryFind`

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

```
let rec tryFind p xs =
  match xs with
  | 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
```

Exercise: union of sets

Let an insertion function be declared by

```
let insert x ys = if isMember x ys then ys else x::ys;;
```

Declare a union function on sets, where a set is represented by a list without duplicated elements.

Remember:

$$\texttt{foldBack}\ (\oplus)\ [x_1;x_2;\ldots;x_n]\ b \rightsquigarrow x_1 \oplus (x_2 \oplus \cdots \oplus (x_n \oplus b)\cdots)$$

Summary

Many recursive list functions follow the same schema. For example:

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

Takeaway:

Succinct declarations achievable using higher-order functions

In other words, avoid repetition by using higher-order functions

More precisely, if you find yourself repeating a pattern, you should try to extract this pattern as a higher-order function

Appendix I

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
  | Square of float
  | Triangle of float*float*float;;
```

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

```
- 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

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

- the union of three disjoint sets

```
type shape =
    Circle of float
  | Square of float
  | Triangle of float*float*float;;
```

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

```
- 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

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

- the union of three disjoint sets

```
type shape =
    Circle of float
  | Square of float
  | Triangle of float*float*float;;
```

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

```
- 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
```

Constructors in Patterns

A shape-area function is declared

```
let area s = match s with
  | 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 -> real
```

following the structure of shapes.

- a constructor only matches itself

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

Constructors in Patterns

A shape-area function is declared

```
let area s = match s with
  | 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 -> real
```

following the structure of shapes.

- a constructor only matches itself

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

Constructors in Patterns

A shape-area function is declared

```
let area s = match s with
  | 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 -> real
```

following the structure of shapes.

- a constructor only matches itself

$$\begin{array}{ll} & \texttt{area (Circle 1.2)} \\ \rightsquigarrow & (\texttt{System.Math.PI * r * r}, [\texttt{r} \mapsto \texttt{1.2}]) \\ \rightsquigarrow & \texttt{...} \end{array}$$

Appendix II

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Aim: A succinct, elegant program reflecting the model.

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Aim: A succinct, elegant program reflecting the model.

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Aim: A succinct, elegant program reflecting the model.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type articleCode = string
type articleName = string
type price       = int
type register    = (articleCode * (articleName*price)) list
type noPieces    = int
type item        = noPieces * articleCode
type purchase    = item list
type info        = noPieces * articleName * price
type infoseq     = info list
type bill        = infoseq * price

makeBill: register -> purchase -> bill
```

Is the model adequate?

A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type articleCode = string
type articleName = string
type price       = int
type register    = (articleCode * (articleName*price)) list
type noPieces    = int
type item        = noPieces * articleCode
type purchase    = item list
type info        = noPieces * articleName * price
type infoseq     = info list
type bill        = infoseq * price

makeBill: register -> purchase -> bill
```

Is the model adequate?

A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type articleCode = string
type articleName = string
type price       = int
type register    = (articleCode * (articleName*price)) list
type noPieces    = int
type item        = noPieces * articleCode
type purchase    = item list
type info        = noPieces * articleName * price
type infoseq     = info list
type bill        = infoseq * price

makeBill: register -> purchase -> bill
```

Is the model adequate?

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill is computed as follows:

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill is computed as follows:

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill is computed as follows:

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Functional decomposition (1)

Type: findArticle: articleCode → register → articleName * price

```
let rec findArticle ac lst = match lst with
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                     -> findArticle ac reg
    | _                          ->
            failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Note that the specified type is an instance of the inferred type.

An article description is found as follows:

```
findArticle "a2" reg;;
val it : string * int = ("herring", 4)

findArticle "a5" reg;;
System.Exception: a5 is an unknown article code
   at FSI_0016.findArticle[a] ...
```

Note: failwith is a built-in function that raises an exception

Functional decomposition (1)

Type: findArticle: articleCode → register → articleName * price

```
let rec findArticle ac lst = match lst with
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                     -> findArticle ac reg
    | _                          ->
          failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Note that the specified type is an instance of the inferred type.

An article description is found as follows:

```
findArticle "a2" reg;;
val it : string * int = ("herring", 4)

findArticle "a5" reg;;
System.Exception: a5 is an unknown article code
   at FSI_0016.findArticle[a] ...
```

Note: failwith is a built-in function that raises an exception

Functional decomposition (2)

Type: makeBill: register → purchase → bill

```
let rec makeBill reg items = match items with
    | []             -> ([],0)
    | (np,ac)::pur ->
          let (aname,aprice) = findArticle ac reg
          let tprice         = np*aprice
          let (billtl,sumtl) = makeBill reg pur
          ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

```
val makeBill :
    (string * ('a * int)) list -> (int * string) list
                 -> (int * 'a * int) list * int
```

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Functional decomposition (2)

Type: makeBill: register → purchase → bill

```
let rec makeBill reg items = match items with
    | []            -> ([],0)
    | (np,ac)::pur ->
          let (aname,aprice) = findArticle ac reg
          let tprice         = np*aprice
          let (billtl,sumtl) = makeBill reg pur
          ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

```
val makeBill :
    (string * ('a * int)) list -> (int * string) list
                -> (int * 'a * int) list * int
```

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Functional decomposition (2)

Type: makeBill: register $\rightarrow$ purchase $\rightarrow$ bill

```
let rec makeBill reg items = match items with
    | []            -> ([],0)
    | (np,ac)::pur ->
          let (aname,aprice) = findArticle ac reg
          let tprice         = np*aprice
          let (billtl,sumtl) = makeBill reg pur
          ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

```
val makeBill :
    (string * ('a * int)) list -> (int * string) list
                -> (int * 'a * int) list * int
```

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Patterns with guards: Three versions of `findArticle`

An if-then-else expression in

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::reg -> if ac=ac' then adesc
                         else findArticle ac reg
   | _   -> failwith(ac + " is an unknown article code");;
```

may be avoided using clauses with guards:

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::reg when ac=ac' -> adesc
   | (ac',adesc)::reg             -> findArticle ac reg
   | _   -> failwith(ac + " is an unknown article code");;
```

This may be simplified using wildcards:

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::_ when ac=ac' -> adesc
   | _::reg                     -> findArticle ac reg
   | _   -> failwith(ac + " is an unknown article code");;
```

Patterns with guards: Three versions of `findArticle`

An if-then-else expression in

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::reg -> if ac=ac' then adesc
                         else findArticle ac reg
   | _   -> failwith(ac + " is an unknown article code");;
```

may be avoided using clauses with guards:

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::reg when ac=ac' -> adesc
   | (ac',adesc)::reg             -> findArticle ac reg
   | _   -> failwith(ac + " is an unknown article code");;
```

This may be simplified using wildcards:

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::_ when ac=ac' -> adesc
   | _::reg                     -> findArticle ac reg
   | _   -> failwith(ac + " is an unknown article code");;
```

Patterns with guards: Three versions of `findArticle`

An if-then-else expression in

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::reg -> if ac=ac' then adesc
                           else findArticle ac reg
   | _    -> failwith(ac + " is an unknown article code");;
```

may be avoided using clauses with guards:

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::reg when ac=ac' -> adesc
   | (ac',adesc)::reg             -> findArticle ac reg
   | _    -> failwith(ac + " is an unknown article code");;
```

This may be simplified using wildcards:

```
let  rec findArticle ac articles = match articles with
   | (ac',adesc)::_ when ac=ac' -> adesc
   | _::reg                     -> findArticle ac reg
   | _    -> failwith(ac + " is an unknown article code");;
```

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.