KSFUPRO1K1U – Functional Programming

Lecture 4: Collections: Lists, maps and sets

Niels Hallenberg

These slides are based on original slides by Michael R. Hansen, DTU. Thanks!!!

The original slides has been used at a course in functional programming at DTU.

Motivation

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 $\ldots$

The collections, List, Maps and Sets share these properties

- expose natural abstract concepts; and may be viewed as *design patterns in the small*.
- efficient code reuse because functions can be used and combined in many ways
- the data structures are *immuttable*.
- the same design patterns are used across the libraries

HIGHER-ORDER FUNCTIONS ARE USEFUL

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
  - 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]
```

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]
```

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

The function: `map`

Applies a function to each element in a list

$$\texttt{map}\ f\ [v_1; v_2; \ldots; v_n] \ = \ [f(v_1); f(v_2); \ldots; f(v_n)]$$

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
```

Does `map` always run through the entire list, assuming *f* returns a value?

Exercise

Declare a function

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

Remember

$$\texttt{map}\ f\ [v_1; v_2; \ldots; v_n] \ = \ [f(v_1); f(v_2); \ldots; f(v_n)]$$

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 = 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
```

Does `exists` always run through the entire list, assuming p returns a value?

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 : $p(x)$.

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

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
```

Does `forall` always run through the entire list, assuming p returns a value?

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

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

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 $p(x) =$ `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 ['1'; 'p'; 'F'; '-'];;
val it : char list = ['p'; 'F']
```

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

Does `filter` always run through the entire list, assuming p returns a value?

Exercise

Declare a function

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

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

Remember:
`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 = 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
```

Does `tryFind` always run through the entire list, assuming *p* returns a value?

Folding a function over a list (I)

Example: sum of norms of geometric vectors:

```
let norm(x1:float,y1:float) = sqrt(x1*x1+y1*y1);;
val norm : float * float -> float
```

```
let rec sumOfNorms = function
    | []     -> 0.0
    | v::vs -> norm v + sumOfNorms vs;;
val sumOfNorms : (float * float) list -> float
```

```
let vs = [(1.0,2.0); (2.0,1.0); (2.0, 5.5)];;
val vs : (float * float) list
       = [(1.0, 2.0); (2.0, 1.0); (2.0, 5.5)]

sumOfNorms vs;;
val it : float = 10.32448591
```

Folding a function over a list (II)

```
let rec sumOfNorms = function
    | []     -> 0.0
    | v::vs -> norm v + sumOfNorms vs;;
```

Let $f\ v\ s$ abbreviate $\texttt{norm}\ v + s$ in the evaluation:

$$
\begin{array}{ll}
 & \texttt{sumOfNorms}\ [v_0; v_1; \ldots; v_{n-1}] \\
\rightsquigarrow & \texttt{norm}\ v_0 + (\texttt{sumOfNorms}\ [v_1; \ldots; v_{n-1}]) \\
= & f\ v_0\ (\texttt{sumOfNorms}\ [v_1; \ldots; v_{n-1}]) \\
\rightsquigarrow & f\ v_0\ (f\ v_1\ (\texttt{sumOfNorms}[v_2; \ldots; v_{n-1}])) \\
\vdots & \\
\rightsquigarrow & f\ v_0\ (f\ v_1\ (\cdots (f\ v_{n-1}\ 0.0)\cdots))
\end{array}
$$

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

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

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

Using the cons operator gives the append function `@` on lists:

$$\texttt{foldBack (fun x rst -> x::rst)}\ [x_0;\ x_1;\ \ldots;\ x_{n-1}]\ \texttt{ys}$$
$$= x_0 \texttt{::} (x_1 \texttt{::} \ \ldots\ \texttt{::} (x_{n-1} \texttt{::ys}) \ \ldots\ ))$$
$$= [x_0;\ x_1;\ \ldots;\ x_{n-1}]\ \texttt{@ ys}$$

so we get:

```
let (@) xs ys = List.foldBack (fun x rst -> x::rst) xs ys;;
val ( @ ) : 'a list -> 'a list -> 'a list

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

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 sumOfNorms vs = foldBack (fun v s -> norm v + s)  vs 0.0;;

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

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

Does `foldBack` always run through the entire list, assuming *f* returns a value?

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:

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

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

Suppose that $\oplus$ is an infix function.

Then the `fold` function has the definitions:

$$\begin{array}{rl} \texttt{fold}\ (\oplus)\ e_a\ [b_0;\ b_1;\ \ldots\ ;\ b_{n-2};\ b_{n-1}\ ] & \\ = & ((\ldots((e_a \oplus b_0) \oplus b_1)\ldots) \oplus b_{n-2}) \oplus b_{n-1} \end{array}$$

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

Examples:

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

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]
```

Summary

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

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

Overview

Sets and Maps as abstract data types

- Useful in the modelling and solution of many problems
- Many similarities with the list library

Recommendation: Use these libraries whenever it is appropriate.

The set concept (1)

A *set* (in mathematics) is a collection of element like

$$\{\text{Bob}, \text{Bill}, \text{Ben}\}, \{1, 3, 5, 7, 9\}, \mathbb{N}, \text{and } \mathbb{R}$$

- the sequence in which elements are enumerated is of no concern, and
- repetitions among members of a set is of no concern either

It is possible to decide whether a given value is in the set.

$$\text{Alice} \notin \{\text{Bob}, \text{Bill}, \text{Ben}\} \qquad \text{and} \qquad 7 \in \{1, 3, 5, 7, 9\}$$

The empty set containing no element is written $\{\}$ or $\emptyset$.

The sets concept (2)

A set A is a *subset* of a set B, written $A \subseteq B$, if all the elements of A are also elements of B, for example

$$\{\mathrm{Ben}, \mathrm{Bob}\} \subseteq \{\mathrm{Bob}, \mathrm{Bill}, \mathrm{Ben}\} \qquad \text{and} \qquad \{1, 3, 5, 7, 9\} \subseteq \mathbb{N}$$

Two sets A and B are equal, if they are both subsets of each other:

$$A = B \qquad \text{if and only if} \qquad A \subseteq B \text{ and } B \subseteq A$$

i.e. two sets are equal if they contain exactly the same elements.

The subset of a set A which consists of those elements satisfying a predicate p can be expressed using a *set-comprehension*:

$$\{x \in A \mid p(x)\}$$

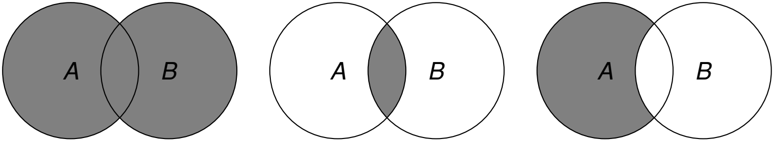
For example:

$$\{1, 3, 5, 7, 9\} = \{x \in \mathbb{N} \mid \mathrm{odd}(x) \text{ and } x < 11\}$$

The set concept (3)

Some standard operations on sets:

$$
\begin{array}{lcll}
A \cup B & = & \{x \mid x \in A \text{ or } x \in B\} & \text{union} \\
A \cap B & = & \{x \mid x \in A \text{ and } x \in B\} & \text{intersection} \\
A \setminus B & = & \{x \in A \mid x \notin B\} & \text{difference}
\end{array}
$$



(a) $A \cup B$ (b) $A \cap B$ (c) $A \setminus B$

Figure: Venn diagrams for (a) union, (b) intersection and (c) difference

For example

$$
\begin{array}{lcl}
\{\text{Bob}, \text{Bill}, \text{Ben}\} \cup \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Alice}, \text{Ann}, \text{Bob}, \text{Bill}, \text{Ben}\} \\
\{\text{Bob}, \text{Bill}, \text{Ben}\} \cap \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Bill}\} \\
\{\text{Bob}, \text{Bill}, \text{Ben}\} \setminus \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Bob}, \text{Ben}\}
\end{array}
$$

Abstract Data Types

An abstract Data Type: A type together with a collection of operations, where

- the representation of values is hidden.

An abstract data type for sets must have:

- Operations to generate sets from the elements. Why?
- Operations to extract the elements of a set. Why?
- Standard operations on sets.

Sets in F#

The `Set` library of F# supports finite sets. An efficient implementation is based on a balanced binary tree.

Examples:

```
set ["Bob"; "Bill"; "Ben"];;
val it : Set<string> = set ["Ben"; "Bill"; "Bob"]

set [3; 1; 9; 5; 7; 9; 1];;
val it : Set<int> = set [1; 3; 5; 7; 9]
```

Equality of two sets is tested in the usual manner:

```
set["Bob";"Bill";"Ben"] = set["Bill";"Ben";"Bill";"Bob"];;
val it : bool = true
```

Sets are ordered on the basis of a lexicographical ordering:

```
compare (set ["Ann";"Jane"]) (set ["Bill";"Ben";"Bob"]);;
val it : int = -1
```

Selected operations (1)

- `ofList: 'a list -> Set<'a>`,
  where `ofList` $[a_0;\ldots;a_{n-1}] = \{a_0;\ldots;a_{n-1}\}$
- `toList: Set<'a> -> 'a list`,
  where `toList` $\{a_0,\ldots,a_{n-1}\} = [a_0;\ldots;a_{n-1}]$
- `add: 'a -> Set<'a> -> Set<'a>`,
  where `add` $a\,A = \{a\} \cup A$
- `remove: 'a -> Set<'a> -> Set<'a>`,
  where `remove` $a\,A = A \setminus \{a\}$
- `contains: 'a -> Set<'a> -> bool`,
  where `contains` $a\,A = a \in A$
- `minElement: Set<'a> -> 'a)`
  where `minElement` $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\} = a_0$ when $n > 0$

Notice that minElement is well-defined due to the ordering:

```
Set.minElement (Set.ofList ["Bob"; "Bill"; "Ben"]);;
val it : string = "Ben"
```

Selected operations (2)

- `union: Set<'a> -> Set<'a> -> Set<'a>`,
  where `union` $A\ B = A \cup B$
- `intersect: Set<'a> -> Set<'a> -> Set<'a>`,
  where `intersect` $A\ B = A \cap B$
- `difference: Set<'a> -> Set<'a> -> Set<'a>`,
  where `difference` $A\ B = A \setminus B$
- `exists: ('a -> bool) -> Set<'a> -> bool`,
  where `exists` $p\ A = \exists x \in A.p(x)$
- `forall: ('a -> bool) -> Set<'a> -> bool`,
  where `forall` $p\ A = \forall x \in A.p(x)$
- `fold: ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a`,
  where

$$\begin{aligned}&\texttt{fold}\ f\ a\ \{b_0, b_1, \ldots, b_{n-2}, b_{n-1}\}\\ &= f(f(f(\cdots f(f(a, b_0), b_1), \ldots), b_{n-2}), b_{n-1})\end{aligned}$$

These work similar to their List siblings, e.g.

`Set.fold (-) 0 (set [1; 2; 3])` $= ((0-1)-2)-3 = -6$

where the ordering is exploited.

Example: Map Coloring (1)

Maps and colors are modelled in a more natural way using sets:

```
type country  = string;;
type map      = Set<country*country>;;
type color    = Set<country>;;
type coloring = Set<color>;;
```

WHY?

Two countries c_1, c_2 are neighbors in a map m,
if either $(c_1, c_2) \in m$ or $(c_2, c_1) \in m$:

```
let areNb c1 c2 m =
   Set.contains (c1,c2) m || Set.contains (c2,c1) m;;
```

Color *col* can be extended by a country c given map m,
if for every country c' in *col*: c and c' are not neighbours in m

```
let canBeExtBy m col c =
   Set.forall (fun c' -> not (areNb c' c m)) col;;
```

Example: Map Coloring (2)

The function

```
extColoring: map -> coloring -> country -> coloring
```

is declared as a recursive function over the coloring:

```
let rec extColoring m cols c =
   if Set.isEmpty cols
   then Set.singleton (Set.singleton c)
   else let col = Set.minElement cols
        let cols' = Set.remove col cols
        if canBeExtBy m col c
        then Set.add (Set.add c col) cols'
        else Set.add col (extColoring m cols' c);;
```

Notice similarity to a list recursion:

- base case [] corresponds to the empty set
- for a recursive case x::xs, the head x corresponds to the minimal element col and the tail xs corresponds to the ”rests” set cols’

Example: Map Coloring (3)

The list-based version, from last lecture:

```
let rec extColoring m cols c =
    match cols with
    | []          -> [[c]]
    | col::cols' -> if canBeExtBy m col c
                    then (c::col)::cols'
                    else col::extColoring m cols' c
```

The set-based version:

```
let rec extColoring m cols c =
   if Set.isEmpty cols
   then Set.singleton (Set.singleton c)
   else let col = Set.minElement cols
        let cols' = Set.remove col cols
        if canBeExtBy m col c
        then Set.add (Set.add c col) cols'
        else Set.add col (extColoring m cols' c)
```

The list-based version is more efficient (why?) and more readable.

Example: Map Coloring (4)

A set of countries is obtained from a map by the function:

```
countries: map -> Set<country>
```

that is based on repeated insertion of the countries into a set:

```
let countries m =
   Set.fold
      (fun set (c1,c2) -> Set.add c1 (Set.add c2 set))
      Set.empty
      m
```

The function

```
colCntrs: map -> Set<country> -> coloring
```

is based on repeated insertion of countries in colorings using the `extColoring` function:

```
let colCntrs m cs = Set.fold (extColoring m) Set.empty cs
```

Type of `Set.fold`:

```
(('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a)
```

Example: Map Coloring (5)

The function that creates a coloring from a map is declared using functional composition:

```
let colMap m = colCntrs m (countries m);;

let exMap = Set.ofList [("a","b"); ("c","d"); ("d","a")];;

colMap exMap;;
val it : Set<Set<string>>
       = set [set ["a"; "c"]; set ["b"; "d"]]
```

The map concept

A *map* from a set A to a set B is a *finite* subset A' of A together with a *function* m defined on A': $m : A' \to B$.
The set A' is called the *domain* of m: $\mathrm{dom}\, m = A'$.

A map m can be described in a tabular form:

a_0	b_0
a_1	b_1
$\vdots$	
a_{n-1}	b_{n-1}

- An element a_i in the set A' is called a *key*
- A pair (a_i, b_i) is called an *entry*, and
- b_i is called the *value* for the key a_i.

We denote the sets of entries of a map as follows:

$$\mathrm{entriesOf}(m) = \{(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})\}$$

Selected map operations in F#

- `ofList: ('a*'b) list -> Map<'a,'b>`
  `ofList` $[(a_0, b_0);\ldots;(a_{n-1}, b_{n-1})] = m$
- `add: 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>`
  `add` $a\ b\ m = m'$, where m' is obtained m by overriding m with the entry (a, b)
- `find: 'a -> Map<'a,'b> -> 'b`
  `find` $a\ m = m(a)$, if $a \in \mathrm{dom}\ m$;
  otherwise an exception is raised
- `tryFind: 'a -> Map<'a,'b> -> 'b option`
  `tryFind` $a\ m =$ `Some` $(m(a))$, if $a \in \mathrm{dom}\ m$; `None` otherwise
- `foldBack: ('a->'b->'c->'c) -> Map<'a,'b> -> 'c -> 'c`
  `foldBack` $f\ m\ c = f\ a_0\ b_0\ (f\ a_1\ b_1\ (f \ldots (f\ a_{n-1}\ b_{n-1}\ c) \cdots))$

A few examples

```
let reg1 = Map.ofList [("a1",("cheese",25));
                        ("a2",("herring",4));
                        ("a3",("soft drink",5))];;
val reg1 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));
       ("a3", ("soft drink", 5))]
```

An entry can be added to a map using `add` and the value for a key in a map is retrieved using either `find` or `tryFind`:

```
let reg2 = Map.add "a4" ("bread", 6) reg1;;
val reg2 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));
       ("a3", ("soft drink", 5)); ("a4", ("bread", 6))]

Map.find "a2" reg1;;
val it : string * int = ("herring", 4)

Map.tryFind "a2" reg1;;
val it : (string * int) option = Some ("herring", 4)
```

An example using Map.foldBack

We can extract the list of article codes and prices for a given register using the fold functions for maps:

```
let reg1 = Map.ofList [("a1",("cheese",25));
                       ("a2",("herring",4));
                       ("a3",("soft drink",5))];;

Map.foldBack (fun ac (_,p) cps -> (ac,p)::cps) reg1 [];;
val it : (string * int) list =
    [("a1", 25); ("a2", 4); ("a3", 5)]
```

This and other higher-order functions are similar to their List and Set siblings.

Example: Cash register (1)

```
type articleCode = string;;
type articleName = string;;
type noPieces    = int;;
type price       = int;;

type info        = noPieces * articleName * price;;
type infoseq     = info list;;
type bill        = infoseq * price;;
```

The natural model of a register is using a map:

```
type register    = Map<articleCode, articleName*price>;;
```

since an article code is *a unique identification* of an article.

First version:

```
type item        = noPieces * articleCode;;
type purchase    = item list;;
```

Example: Cash register (1) - a recursive program

```
exception FindArticle;;

(* makebill: register -> purchase -> bill *)
let rec makeBill reg = function
    | []              -> ([],0)
    | (np,ac)::pur ->
        match Map.tryFind ac reg with
        | None                  -> raise FindArticle
        | Some(aname,aprice) ->
            let tprice            = np*aprice
            let (infos,sumbill) = makeBill reg pur
            ((np,aname,tprice)::infos, tprice+sumbill);;

let pur = [(3,"a2"); (1,"a1")];;
makeBill reg1 pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

- the lookup in the register is managed by a `Map.tryFind`

Example: Cash register (2) - using List.foldBack

```
let makeBill' reg pur =
   let f (np,ac) (infos,billprice)
          = let (aname, aprice) = Map.find ac reg
            let tprice          = np*aprice
            ((np,aname,tprice)::infos, tprice+billprice)
   List.foldBack f pur ([],0);;

makeBill' reg1 pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

- the recursion is handled by List.foldBack
- the exception is handled by `Map.find`

Example: Cash register (2) - using maps for purchases

The purchase: 3 herrings, one piece of cheese, and 2 herrings, is the same as a purchase of one piece of cheese and 5 herrings.

A purchase associated number of pieces with article codes:

```
type purchase     = Map<articleCode,noPieces>;;
```

A bill is produced by folding a function over a map-purchase:

```
let makeBill'' reg pur =
   let f ac np (infos,billprice)
         = let (aname, aprice) = Map.find ac reg
           let tprice          = np*aprice
           ((np,aname,tprice)::infos, tprice+billprice)
   Map.foldBack f pur ([],0);;

let purMap = Map.ofList [("a2",3); ("a1",1)];;
val purMap : Map<string,int> = map [("a1", 1); ("a2", 3)]

makeBill'' reg1 purMap;;
val it = ([(1, "cheese", 25); (3, "herring", 12)], 37)
```

Summary

- The concepts of sets and maps.
- Fundamental operations on sets and maps.
- Applications of sets and maps.