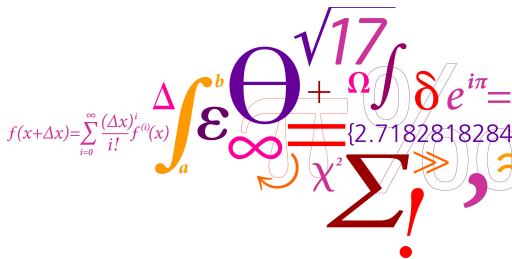


02157 Functional Programming

Collections: Finite Sets and Maps

Michael R. Hansen



DTU Compute

Department of Applied Mathematics and Computer Science

- Conventions and tradition
 - example with higher-order list functions

- Conventions and tradition
 - example with higher-order list functions
- Sets and Maps as abstract data types
 - Useful when modelling and when programming
 - Many similarities with the list library

Conventions and tradition

exemplified using the Flights and Luggage problem
from Week 4
– an old exam question

Conventions and tradition

- type names are descriptive and start with capital letters

exemplified using the Flights and Luggage problem
from Week 4
– an old exam question

Conventions and tradition

- type names are descriptive and start with capital letters
- function names are descriptive and start with small letters

exemplified using the Flights and Luggage problem
from Week 4
– an old exam question

Conventions and tradition

- type names are descriptive and start with capital letters
- function names are descriptive and start with small letters
- variable names are short and consistently used

exemplified using the Flights and Luggage problem
from Week 4
– an old exam question

Conventions and tradition

- type names are descriptive and start with capital letters
- function names are descriptive and start with small letters
- variable names are short and consistently used
- function types are stated in comments

exemplified using the Flights and Luggage problem
from Week 4
– an old exam question

Conventions and tradition

- type names are descriptive and start with capital letters
- function names are descriptive and start with small letters
- variable names are short and consistently used
- function types are stated in comments
- types are avoided in function declarations – unless needed

exemplified using the Flights and Luggage problem
from Week 4
– an old exam question

Conventions and tradition

- type names are descriptive and start with capital letters
- function names are descriptive and start with small letters
- variable names are short and consistently used
- function types are stated in comments
- types are avoided in function declarations – unless needed
- self-check that wanted type is an instance of inferred type
-

exemplified using the Flights and Luggage problem
from Week 4
– an old exam question

Example: Flights and Luggage (1)

- Type names are descriptive and
- start with capital letters

```
type Lid      = string
type Flight   = string
type Airport  = string
```

```
type Route    = (Flight * Airport) list
```

```
type LuggageCatalogue = (Lid * Route) list
```

```
let lcl =
  [ ("DL 016-914",
    [ ("DL 189", "ALT"); ("DL 124", "BRU"); ("SN 733", "CPH") ]
  , ("SK 222-142",
    [ ("SK 208", "ALT"); ("DL 124", "BRU"); ("SK 122", "JFK") ] ) ]
```

Example: Flights and Luggage (2)

- function names are descriptive and start with small letters
- variable names are short and consistently used
- function types are stated in comments
- types are avoided in function declarations – unless needed
- self-check that wanted type is an instance of inferred type

```
// Lid * LuggageCatalogue -> Route
let rec findRoute(lid, lc) =
  match lc with
  | (lid1,r1)::_ when lid1=lid -> r1
  | _::lcrest          -> findRoute(lid,lcrest)
  | _                  -> failwith (lid + " is not found")
```

Example: Flights and Luggage (2)

- function names are descriptive and start with small letters
- variable names are short and consistently used
- function types are stated in comments
- types are avoided in function declarations – unless needed
- self-check that wanted type is an instance of inferred type

```
// Lid * LuggageCatalogue -> Route
let rec findRoute(lid, lc) =
  match lc with
  | (lid1,r1)::_ when lid1=lid -> r1
  | _::lcrest          -> findRoute(lid,lcrest)
  | _                  -> failwith (lid + " is not found")
```

Notice that further comments are not needed and that

- lid, lid', lid1 denote luggage identifiers
- f, f', f1, ... denote flights
- lc, lc', lc1, lcrest denote luggage catalogues
- r, r', r1, .. denote routes

Example: Flights and Luggage (3)

```
// inRoute: Flight -> Route -> bool
let rec inRoute f =
  function
  | (f1,_)::r -> f=f1 || inRoute f r
  | []         -> false
```

Example: Flights and Luggage (3)

```
// inRoute: Flight -> Route -> bool
let rec inRoute f =
  function
  | (f1,_)::r -> f=f1 || inRoute f r
  | []        -> false
```

`List.exists` is the natural choice

– the existence of an element with a certain property is questioned

```
let inRoute f r = List.exists (fun (f1,_) -> f=f1) r
```

Example: Flights and Luggage (4)

- the function `inRoute` is handy to use

```
// withFlight: Flight -> LuggageCatalogue -> Lid list
let rec withFlight f =
  function
  | [] -> []
  | (lid,r)::lc when inRoute f r -> lid::withFlight f lc
  | _::lc -> withFlight f lc
```

Example: Flights and Luggage (4)

- the function `inRoute` is handy to use

```
// withFlight: Flight -> LuggageCatalogue -> List
let rec withFlight f =
  function
  | [] -> []
  | (lid,r)::lc when inRoute f r -> lid::withFlight f lc
  | _::lc -> withFlight f lc
```

- It is natural to use a fold function due to the "non-trivial" processing of elements in luggage catalogues.
- Since there is no restriction on the sequence of the luggage identifiers, `fold` and `foldBack` are both natural choices.

```
let withFlight f lc =
  List.fold
    (fun lids (lid,r) -> if inRoute f r
                        then lid::lids else lids)
    []
    lc
```

Example: Flights and Luggage (5)

```
type ArrivalCatalogue = (Airport * Lid list) list
```

The task:

```
extend : Lid * Route * ArrivalCatalogue → ArrivalCatalogue
```

Example: Flights and Luggage (5)

```
type ArrivalCatalogue = (Airport * Lid list) list
```

The task:

`extend : Lid * Route * ArrivalCatalogue → ArrivalCatalogue`

Invent a function to make things easy

```
// insert: Lid -> Airport
           -> ArrivalCatalogue->ArrivalCatalogue
let rec insert lid airp =
  function
  | []                -> [(airp, [lid])]
  | (airp1, ls)::rest when airp1=airp
                        -> (airp1, lid::ls)::rest
  | pair::rest        -> pair::insert lid airp rest
```

Example: Flights and Luggage (5)

```
type ArrivalCatalogue = (Airport * Lid list) list
```

The task:

$\text{extend} : \text{Lid} * \text{Route} * \text{ArrivalCatalogue} \rightarrow \text{ArrivalCatalogue}$

Invent a function to make things easy

```
// insert: Lid -> Airport
           -> ArrivalCatalogue->ArrivalCatalogue
let rec insert lid airp =
  function
  | []                -> [(airp, [lid])]
  | (airp1, ls)::rest when airp1=airp
                        -> (airp1, lid::ls)::rest
  | pair::rest        -> pair::insert lid airp rest

let rec extend(lid, r, ac) =
  match r with
  | []                -> ac
  | (f, airp)::r1    -> extend (lid, r1, insert lid airp ac)
```

Example: Flights and Luggage (6)

```
type LuggageCatalogue = (Lid * Route) list
type ArrivalCatalogue = (Airport * Lid list) list
```

The task:

$\text{toArrivalCatalogue} : \text{LuggageCatalogue} \rightarrow \text{ArrivalCatalogue}$

should be solved using a fold function and

$\text{extend} : \text{Lid} * \text{Route} * \text{ArrivalCatalogue} \rightarrow \text{ArrivalCatalogue}$

Example: Flights and Luggage (6)

```
type LuggageCatalogue = (Lid * Route) list
type ArrivalCatalogue = (Airport * Lid list) list
```

The task:

$\text{toArrivalCatalogue} : \text{LuggageCatalogue} \rightarrow \text{ArrivalCatalogue}$

should be solved using a fold function and

$\text{extend} : \text{Lid} * \text{Route} * \text{ArrivalCatalogue} \rightarrow \text{ArrivalCatalogue}$

```
let toArrivalCatalogue lc =
  List.foldBack (fun (lid,r) ac -> extend(lid,r,ac)) lc []
```

Example: Flights and Luggage (6)

```
type LuggageCatalogue = (Lid * Route) list
type ArrivalCatalogue = (Airport * Lid list) list
```

The task:

$\text{toArrivalCatalogue} : \text{LuggageCatalogue} \rightarrow \text{ArrivalCatalogue}$

should be solved using a fold function and

$\text{extend} : \text{Lid} * \text{Route} * \text{ArrivalCatalogue} \rightarrow \text{ArrivalCatalogue}$

```
let toArrivalCatalogue lc =
  List.foldBack (fun (lid,r) ac -> extend(lid,r,ac)) lc []
```

Naming conventions and tradition support readability

Example: Flights and Luggage (6)

```
type LuggageCatalogue = (Lid * Route) list
type ArrivalCatalogue = (Airport * Lid list) list
```

The task:

`toArrivalCatalogue : LuggageCatalogue → ArrivalCatalogue`

should be solved using a fold function and

`extend : Lid * Route * ArrivalCatalogue → ArrivalCatalogue`

```
let toArrivalCatalogue lc =
  List.foldBack (fun (lid,r) ac -> extend(lid,r,ac)) lc []
```

Naming conventions and tradition support readability

– and so does functional decomposition

FSharp's immutable collections

Sets and Maps

- List: **a finite** sequence of elements of the same type
 - the sequence in which elements are enumerated is important
 - repetitions among elements of a list matters

FSharp's **immutable** collections

- List: a **finite** sequence of elements of the same type
 - the sequence in which elements are enumerated is important
 - repetitions among elements of a list matters
- Set: a **finite** collection of elements of the same type
 - the sequence in which elements are enumerated is of no concern
 - repetitions among members of a set is of no concern

Today

FSharp's **immutable** collections

- List: a **finite** sequence of elements of the same type
 - the sequence in which elements are enumerated is important
 - repetitions among elements of a list matters
- Set: a **finite** collection of elements of the same type
 - the sequence in which elements are enumerated is of no concern
 - repetitions among members of a set is of no concern

Today

- Map: a **finite** function from a domain of keys to values
 - the uniqueness of keys is an important property

Today

- List: a **finite** sequence of elements of the same type
 - the sequence in which elements are enumerated is important
 - repetitions among elements of a list matters
- Set: a **finite** collection of elements of the same type
 - the sequence in which elements are enumerated is of no concern
 - repetitions among members of a set is of no concern

Today

- Map: a **finite** function from a domain of keys to values
 - the uniqueness of keys is an important property

Today

- Sequence: a **possibly infinite** sequence of elements of the same type
 - the elements of a sequence are computed by demand

Covered later in the semester

Types, data types and abstract data types

- A **type** is generated from basic types
`int, float, bool, string, ...` and type variables
`'a, 'b, 'c, ...` using type operators `*`, `->`, `list`, `...`

Types, data types and abstract data types

- A **type** is generated from basic types

`int, float, bool, string, ...` and type variables

`'a, 'b, 'c, ...` using type operators `*`, `->`, `list`, `...`

- A **data type** is characterized by

- a type
- a set of values
- a set of operations

`'alist`
`[ ], [v], [v1; ... vn]`
`::, @, List.rev, List.fold, ...`

Types, data types and abstract data types

- A **type** is generated from basic types
`int, float, bool, string, ...` and type variables
`'a, 'b, 'c, ...` using type operators `*`, `->`, `list`, ...
- A **data type** is characterized by
 - a type
 - a set of values
 - a set of operations

`'alist`
`[ ], [v], [v1; ... vn]`
`::, @, List.rev, List.fold, ...`
- A **abstract data type** is a data type
 - where the representation of values is **hidden** LiskovZilles 1974

Types, data types and abstract data types

- A **type** is generated from basic types
`int, float, bool, string, ...` and type variables
`'a, 'b, 'c, ...` using type operators `*`, `->`, `list`, ...
- A **data type** is characterized by
 - a type
 - a set of values
 - a set of operations

`'alist`
`[ ], [v], [v1; ... vn]`
`::, @, List.rev, List.fold, ...`
- A **abstract data type** is a data type
 - where the representation of values is **hidden** LiskovZilles 1974

Examples:

- `List` is a data type but not an abstract one
 — the representation of list values is visible (`[]` and `::`)
- `Set` and `Map` are abstract data types

The set concept (1)

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

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

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

The set concept (1)

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

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

- 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.

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

The set concept (1)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

The set concept (3)

Some standard operations on sets:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

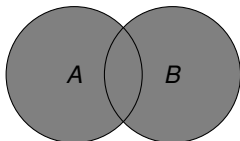
$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

$$A \setminus B = \{x \in A \mid x \notin B\}$$

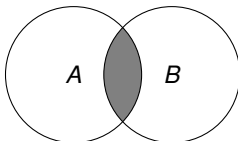
union

intersection

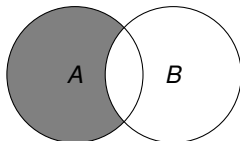
difference



(a) $A \cup B$



(b) $A \cap B$



(c) $A \setminus B$

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

The set concept (3)

Some standard operations on sets:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

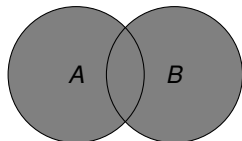
$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

$$A \setminus B = \{x \in A \mid x \notin B\}$$

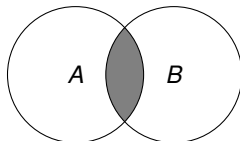
union

intersection

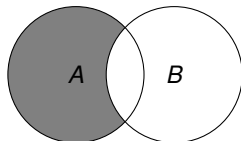
difference



(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

$$\{\text{Bob, Bill, Ben}\} \cup \{\text{Alice, Bill, Ann}\} = \{\text{Alice, Ann, Bob, Bill, Ben}\}$$

$$\{\text{Bob, Bill, Ben}\} \cap \{\text{Alice, Bill, Ann}\} = \{\text{Bill}\}$$

$$\{\text{Bob, Bill, Ben}\} \setminus \{\text{Alice, Bill, Ann}\} = \{\text{Bob, Ben}\}$$

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

- the representation of values is hidden.

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.

Immutable Sets in F#

An Abstract Data Type: `Set<'a>`

An abstract type for sets should at least support the following:

```
empty:      Set<'a>
add:        'a -> Set<'a> -> Set<'a>
union:      Set<'a> -> Set<'a> -> Set<'a>
intersect:  Set<'a> -> Set<'a> -> Set<'a>
difference: Set<'a> -> Set<'a> -> Set<'a>
contains:   'a -> Set<'a> -> bool
toList:     Set<'a> -> 'a list
```

where

- any finite set can be generated by repeatedly **adding** elements to the **empty** set;
- **union**, **intersection** and **difference** are fundamental set operations;
- **contains** and **toList** are used to inspect the set

An Abstract Data Type: `Set<'a>`

An abstract type for sets should at least support the following:

```
empty:      Set<'a>
add:        'a -> Set<'a> -> Set<'a>
union:      Set<'a> -> Set<'a> -> Set<'a>
intersect:  Set<'a> -> Set<'a> -> Set<'a>
difference: Set<'a> -> Set<'a> -> Set<'a>
contains:   'a -> Set<'a> -> bool
toList:     Set<'a> -> 'a list
```

where

- any finite set can be generated by repeatedly **adding** elements to the **empty** set;
- **union**, **intersection** and **difference** are fundamental set operations;
- **contains** and **toList** are used to inspect the set

Note:

- the above operations are supported by the library `Set`.
- the representation of sets used by `Set` is hidden from the user.

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

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
```

Immutability of `Set<'a>`

```
let s = Set.ofList [3; 2; 0];;  
val s : Set<int> = set [0; 2; 3]  
  
Set.add 1 s;  
val it : Set<int> = set [0; 1; 2; 3]  
  
s;  
val it : Set<int> = set [0; 2; 3]
```

Evaluation of `Set.add 1 s` does not change the value of `s`.

Selected further operations (1)

- `ofList`: `'a list -> Set<'a>`,
where `ofList [a0;...;an-1] = {a0;...;an-1}`
- `remove`: `'a -> Set<'a> -> Set<'a>`,
where `remove a A = A \ {a}`
- `minElement`: `Set<'a> -> 'a`
where `minElement {a0, a1, ..., an-2, an-1} = a0` when $n > 0$
(assuming that the enumeration respect the ordering)

Selected further operations (1)

- `ofList`: `'a list -> Set<'a>`,
where `ofList [a0;...;an-1] = {a0;...;an-1}`
- `remove`: `'a -> Set<'a> -> Set<'a>`,
where `remove a A = A \ {a}`
- `minElement`: `Set<'a> -> 'a`
where `minElement {a0, a1, ..., an-2, an-1} = a0` when $n > 0$
(assuming that the enumeration respect the ordering)

Notice that `minElement` on a non-empty set is well-defined due to the ordering:

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

Selected further operations (2)

- `filter`: $('a \rightarrow \text{bool}) \rightarrow \text{Set} <'a> \rightarrow \text{Set} <'a>$, where
 $\text{filter } p \ A = \{x \in A \mid p(x)\}$
- `exists`: $('a \rightarrow \text{bool}) \rightarrow \text{Set} <'a> \rightarrow \text{bool}$,
 where $\text{exists } p \ A = \exists x \in A. p(x)$
- `forall`: $('a \rightarrow \text{bool}) \rightarrow \text{Set} <'a> \rightarrow \text{bool}$,
 where $\text{forall } p \ A = \forall x \in A. p(x)$
- `fold`: $('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow \text{Set} <'b> \rightarrow 'a$,
 where

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

Selected further operations (2)

- `filter`: $('a \rightarrow \text{bool}) \rightarrow \text{Set} <'a> \rightarrow \text{Set} <'a>$, where
 $\text{filter } p \ A = \{x \in A \mid p(x)\}$
- `exists`: $('a \rightarrow \text{bool}) \rightarrow \text{Set} <'a> \rightarrow \text{bool}$,
 where $\text{exists } p \ A = \exists x \in A. p(x)$
- `forall`: $('a \rightarrow \text{bool}) \rightarrow \text{Set} <'a> \rightarrow \text{bool}$,
 where $\text{forall } p \ A = \forall x \in A. p(x)$
- `fold`: $('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow \text{Set} <'b> \rightarrow 'a$,
 where

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

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

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

where the ordering is exploited.

Example: Map Coloring (1)

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

```
type Country = string;;  
type Map     = Set<Country*Country>;;  
type Color   = Set<Country>;;  
type Coloring = Set<Color>;;
```

WHY?

Example: Map Coloring (1)

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

```
type Country = string;;  
type Map     = Set<Country*Country>;;  
type Color   = Set<Country>;;  
type Coloring = Set<Color>;;
```

WHY?

- repetition of elements is of no concern
- order of elements is of no concern

Example: Map Coloring (1)

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

```
type Country = string;;  
type Map     = Set<Country*Country>;;  
type Color   = Set<Country>;;  
type Coloring = Set<Color>;;
```

WHY?

- repetition of elements is of no concern
- order of elements is of no concern

Function declarations will reveal the adequacy of the model.

Example: Map Coloring (2)

```
type Country = string;;  
type Map      = Set<Country*Country>;;
```

The function:

```
areNb: Country -> Country -> Map -> bool
```

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 = ?
```

Remember:

```
contains: 'a -> Set<'a> -> bool  
exists   : ('a -> bool) -> Set<'a> -> bool
```

Example: Map Coloring (3)

```
type Country = string;;  
type Map     = Set<Country*Country>;;  
type Color   = Set<Country>;;
```

The function

```
canBeExtBy: Map -> Color -> Country -> bool
```

Color col and 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 = ?
```

Example: Map Coloring (3)

```
type Country = string;;  
type Map     = Set<Country*Country>;;  
type Color   = Set<Country>;;
```

The function

```
canBeExtBy: Map -> Color -> Country -> bool
```

Color col and 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 = ?
```

Remember

```
forall: ('a -> bool) -> Set<'a> -> bool
```

Example: Map Coloring (4)

```
type Coloring = Set<Color>;;
```

The function

```
extColoring: Map -> Coloring -> Country -> Coloring
```

is declared as a recursive function over the coloring:

WHY?

Example: Map Coloring (4)

```
type Coloring = Set<Color>;;
```

The function

```
extColoring: Map -> Coloring -> Country -> Coloring
```

is declared as a recursive function over the coloring:

WHY?

```
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);;
```

Example: Map Coloring (4)

```
type Coloring = Set<Color>;;
```

The function

```
extColoring: Map -> Coloring -> Country -> Coloring
```

is declared as a recursive function over the coloring:

WHY?

```
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);;
```

Observations

- **Ugly** compared to list version where pattern matching is used
- List version is **more efficient**

Example: Map Coloring (5)

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>;;      // Color list is better
```

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 = ?
```

Example: Map Coloring (5)

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>;;      // Color list is better
```

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 = ?
```

Remember

```
fold:      ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a
foldBack:  ('a -> 'b -> 'b) -> Set<'a> -> 'b -> 'b
Set.add:   'a -> Set<'a> -> Set<'a>
```

Example: Map Coloring (6)

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>;;      // Color list is better
```

The function

```
colCntrs: Map -> Set<Country> -> Coloring
```

is based on repeated extension of colorings by countries using the `extColoring` function:

```
let colCntrs m cs = ?
```

Remember

```
fold:      ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a
foldBack: ('a -> 'b -> 'b) -> Set<'a> -> 'b -> 'b
```

```
extColoring: Map -> Coloring -> Country -> Coloring
```

Example: Map Coloring (7)

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"]]
```

Immutable Maps in F#

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' : $m : A' \rightarrow B$.

The set A' is called the *domain* of m : $\text{dom } m = A'$.

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:

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

Keys are unique

Selected map operations in F#

- `ofList: ('a*'b) list -> Map<'a,'b>`
`ofList [(a0, b0); ...; (an-1, bn-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 ∈ dom m`;
 otherwise an exception is raised
- `tryFind: 'a -> Map<'a,'b> -> 'b option`
`tryFind a m = Some (m(a))`, if `a ∈ dom m`; `None` otherwise
- `foldBack: ('a->'b->'c->'c) -> Map<'a,'b> -> 'c -> 'c`
`foldBack f m c = f a0 b0 (f a1 b1 (f ... (f an-1 bn-1 c) ... ))`

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;;
```

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;;
```

An example

```
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)
```

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)
```

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)
```

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

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

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