λ

# ML

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

# Next lectures

- Thursday May 23: exam simulation
- Tuesday May 28: last lecture
  - We will have a lab class: bring your laptop
  - We will see results and a solution of the ML mini-challenge

# Today


- Recap
- User-defined types
- Signatures and structures


# Recap

# Curried functions

- Functions in ML have only one argument
- Functions with two arguments can be implemented as
  - A function with a tuple as argument
  - Curried form
    - Unary function takes argument $x$
    - The result is a function $f(x)$ that takes argument $y$

- Curried function: divides its arguments such that they can be partially supplied producing intermediate functions that accept the remaining arguments (from Haskell Curry)

# Partial instantiation

- Curried functions are useful because they allow us to create partially instantiated or specialized functions where some (but not all) arguments are supplied.

```
> fun exponent2 x 0 = 1.0
    | exponent2 x y = x * exponent2 x (y-1);
val exponent2 = fn: real -> int -> real

> val g = exponent2 3.0;
val g = fn: int -> real

> g 4;
val it = 81.0: real

> g (4);
val it = 81.0: real
```

We are partially instantiating exponent2 (with name g) – g is the power function with base 3.0

# Another example

```
> val sorted3 = fn x => fn y => fn z =>
                z>=y andalso y>=x
val t1 = (((sorted3 7)9)11
```

- Calling `sorted3 7` returns a function (`fn y => fn z =>`
`z>=y andalso y>=x`)
- Calling it on 9 returns a function `fn z => z>=y andalso y>=x`
- Calling it on 11 returns `true`

# Function application and curried function

- Function application is left-associative

`e1 e2 e3 e4` means `(((e1 e2) e3) e4 )`

- We can call our function simply using spaces rather than tuples:

```
val t1 = sorted3 7 9 11
```

- We can simply write our function as

`fun f p1 p2 p3 … = e` means

```
Val f = fn p1 => fn p2 => fn p3 … => e
```

# `map` function

- The map function accepts two parameters: a function and a list of objects.
- It applies the given function to each object in the list.
- Example:

```
> map (fn x => x + 2) [1,2,3];
val it = [3, 4, 5]: int list
```

# Folding lists: `foldr` and `foldl`

- Similar to the `map` function, but instead of producing a list of values they only produce a single output value.
  - The `foldr` function folds a list of values into a single value starting from the rightmost element

  `> foldr f c [x1, ..., xn]` means $f(x1, f(x2, \dots f(xn, c) \dots)$

  it starts at the rightmost $xn$ with the initial value c

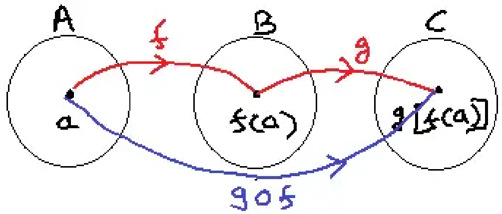
  ```
  > foldr (fn (a,b) => a+b) 2 [1,2,3]
  val it = 8: int
  ```

  - The `foldl` function folds a list of values into a single value starting from the leftmost element

  `> foldl f c [x1, ..., xn]` means $f(xn, f(xn-1, \dots f(x1, c) \dots)$

  it starts at the leftmost $x1$ with the initial value $c$

  ```
  > foldl (fn (a,b) => a+b) 2 [1,2,3]
  val it = 8: int
  ```




# Function composition

# Function composition

- Composition of $F$ and $G$ is the function $H$ such that $H(x) = G(F(x))$
- Example:
  - $F(x) = x + 3$ and $G(y) = y^2 + 2y$,
  - $G\big(F(x)\big) = x^2 + 6x + 9 + 2x + 6 = x^2 + 8x + 15$

# In ML

```
> fun comp (F,G,x) = G(F(x));
val comp = fn: ('a -> 'b) * ('b -> 'c) * 'a ->
'c
> comp (fn x=> x+3, fn y=>y*y+2*y, 10);
val it = 195: int
```

# The operator o

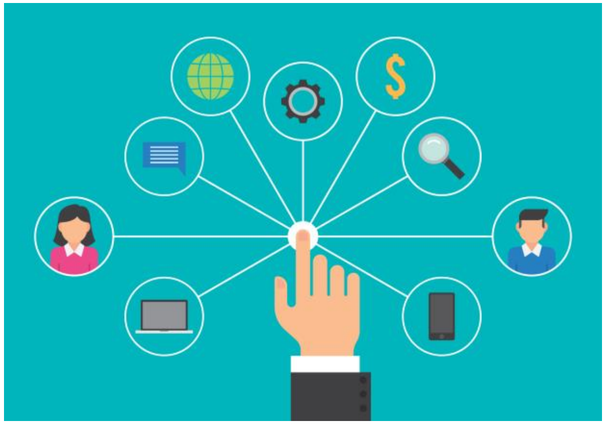
```
> fun F x = x+3;
val F = fn: int -> int

> fun G y = y*y + 2*y;
val G = fn: int -> int

> val H = G o F;
val H = fn: int -> int

> H 10;
val it = 195: int
```

# Exercise L8.9

- In the following exercise, use `map`, `foldr` and `foldl`
  - Define the function `implode`, i.e., `implode[#”b”,#”c”] = “bc”`

# Solution exercise L8.9

```
> val implode = (foldr (op ^) "") o (map str);
val implode = fn: char list -> string
> implode [#”a”,#”b”];
val it = "ab": string
```



# User-defined types

# ML types

- So far
  - `int, real, string, char, bool, unit, exn, instream, outstream`
  - `T1 * T2 * ... * Tn`
  - `T1->T2`
  - `T1 list`
  - `T1 option`
- We can also
  - Rename types
  - Define new types

Type

# (Parameterized) type renaming

# Abbreviations

- Keyword `type`

```
> type signal = int list;
type signal = int list

> val v = [1,2]: signal;
val v = [1, 2]: signal
```

- This is just an abbreviation. If we write

```
> val w = [1,2];
val w = [1, 2]: int list
```

we can then test

```
> v=w;
val it = true: bool
```

# Parametrized type definitions

- In ML we can also parameterize a type definition
- Given two types `'a` and `'b` we declare `mapping` to be a type of lists of pairs of these two types

```
> type ('c,'d) mapping = ('c * 'd) list;
type ('a, 'b) mapping = ('a * 'b) list
```

Note that the type variable names are unimportant

- Example of use of this type

```
> val words = [("in",6),("a",1)] : (string,int) mapping;
val words = [("in", 6), ("a", 1)]: (string, int) mapping
```

# Exercise L9.1

- Give a type definition for a set of sets, where the type of elements is unspecified, and sets are represented by lists

# Solution exercise L9.1

```
> type 'a setOfSets = 'a list list;
type 'a setOfSets = 'a list list
```

# Exercise L9.2

- Give a type definition for a list of triples, the first two components of which have the same type, and the third is some (possibly) different type

# Solution exercise L9.2

```
> type ('a,'b) tripleList = ('a * 'a * 'b) list;
type ('a, 'b) tripleList = ('a * 'a * 'b) list
```

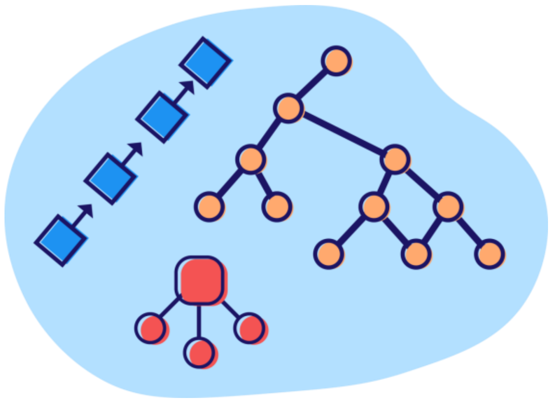
# Exercise L9.3

- Give an example of a value of type `(real,real) mapping`

```
> type ('c,'d) mapping = ('c * 'd) list;
type ('a, 'b) mapping = ('a * 'b) list
```

# Solution exercise L9.3

```
> val x = [(1.0,1.0),(1.0,1.1),(1.1,1.0)] : (real, real)
mapping;

val x = [(1.0, 1.0), (1.0, 1.1), (1.1, 1.0)]: (real, real)
mapping
```



# Datatypes

# Datatypes

- Unlike type declarations, `datatype` creates new types
- Two parts
  - Type constructor, the name of the datatype
  - Data constructors, the possible values
- Example

```
> datatype fruit = Apple | Pear | Grape;
datatype fruit = Apple | Grape | Pear
```

# Use of datatypes

```
> fun isApple (x) = (x = Apple);
val isApple = fn: fruit -> bool
> isApple (Pear);
val it = false: bool
> isApple(Apple);
val it = true: bool
> isApple (Cherry);
poly: : error: Value or constructor (Cherry) has not been
declared
Found near isApple (Cherry)
```

# More general form of datatype definitions

- Type variables can be used to parameterize the datatype
- The data constructors can take arguments (constructor expressions)

```
datatype (<parameters>) <identifier> =
        <first constructor expression> |
        ...
         <last constructor expression>
```

# Constructor expressions and type variables

- Constructor expressions are data constructors that can be parameterized, e.g.,

```
Cherry of int
```

- Any expression of the form `Cherry(i)` is allowed. e.g., `Cherry (23)`
- We can also use type variables instead of `int`, e.g., `Cherry of ‘a`
- Data constructors are used to build expressions that are values for the types
- We can use these types for having types as union types, e.g.,
  - First component, type `’a`
  - Second component, if it exists, of type `’b`

# Unions

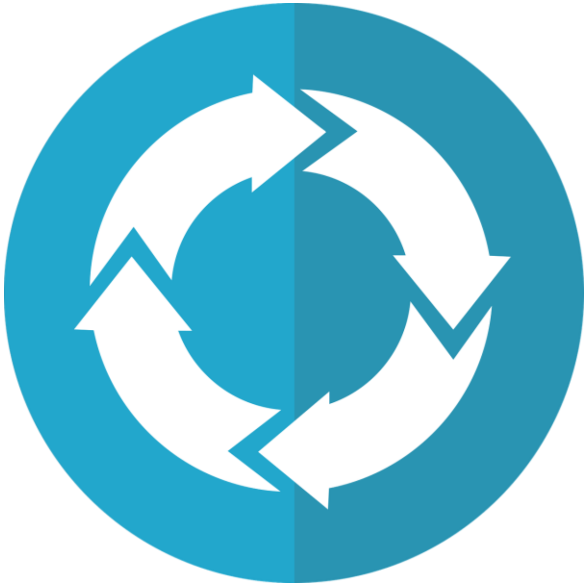
- We can define a type `element` that can be a pair (‘a*’b) or a single (‘a)

```
> datatype ('a,'b) element =
    P of 'a * 'b |
    S of 'a;
datatype ('a, 'b) element = P of 'a * 'b | S of 'a
> P ("a",1);
val it = P ("a", 1): (string, int) element
> P(1.0,2.0);
val it = P (1.0, 2.0): (real, real) element
> S(["a","b"]);
val it = S ["a", "b"]: (string list, 'a) element
```

# Example

- Given a list of `(string,int) element`'s, sum the integers in the second components, when these exist

```
> fun sumElList (nil) = 0
| sumElList (S(x)::L) = sumElList (L)
| sumElList (P(x,y)::L) = y + sumElList (L);
val sumElList = fn: ('a, int) element list ->
int

> sumElList [ P("in",6), S("function"),
P("as",2)];
val it = 8: int
```



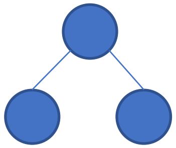
# Recursively defined datatypes

# Recursively defined datatypes

- Binary tree:
  - Empty, or
  - Two children, each of which is, in turn, a binary tree

```
> datatype 'label btree =
    Empty |
    Node of 'label * 'label btree * 'label btree;
datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

btree



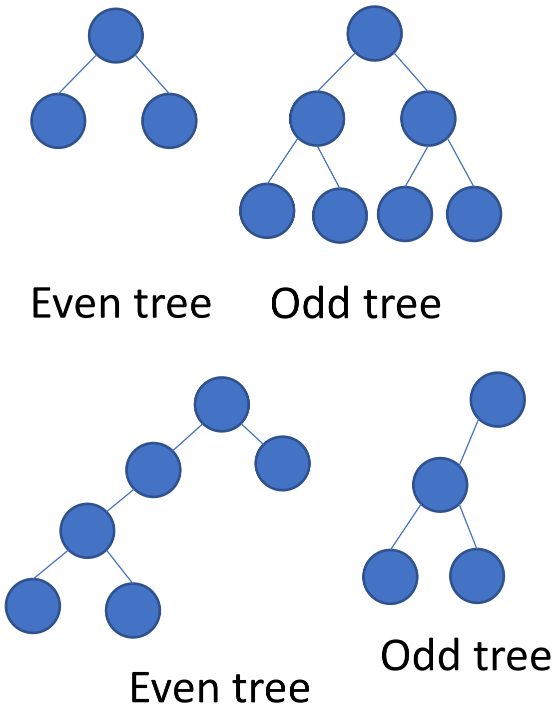
# Example of data

```
> Node ("ML",
        Node ("as",
                Node ("a",Empty,Empty),
                Node ("in",Empty,Empty)
        ),
        Node ("types",Empty,Empty)
);

val it =
        Node
                ("ML", Node ("as", Node ("a", Empty, Empty), Node
("in", Empty, Empty)),
        Node ("types", Empty, Empty)): string btree
```

# Mutually recursive datatypes

- Keyword `and` as with functions
- Example: Even binary trees
  - Even tree: each path from the root to a node with one or two empty subtrees has an even number of nodes
  - Odd tree is defined similarly

- Simple way to define it:
  - Basis: the empty tree is an even tree
  - Induction: a node with a label and two subtrees that are odd trees is the root of an even tree

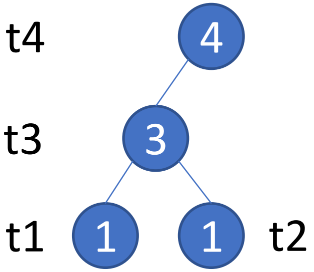


# Example

```
> datatype
        'label evenTree = Empty
                | Enode of 'label * 'label oddTree * 'label oddTree
        and
        'label oddTree =
                Onode of 'label * 'label evenTree * 'label evenTree;
datatype 'a evenTree = Empty | Enode of 'a * 'a oddTree * 'a oddTree
datatype 'a oddTree = Onode of 'a * 'a evenTree * 'a evenTree
```

# Example

```
> val t1 = Onode (1,Empty,Empty);
val t1 = Onode (1, Empty, Empty): int oddTree
> val t2 = Onode (1,Empty,Empty);
val t2 = Onode (1, Empty, Empty): int oddTree
> val t3 = Enode (3,t1,t2);
val t3 = Enode (3, Onode (1, Empty, Empty), Onode (1, Empty,
      Empty)): int evenTree
> val t4 = Onode (4,t3,Empty);
val t4 =
      Onode
      (4, Enode (3, Onode (1, Empty, Empty), Onode (1, Empty,
Empty)), Empty): int oddTree
```



# Signatures and structures

# Signatures and structures

- Structure: sequence of declarations comprising the components of the structure
  - The components of a structure are accessed using long identifiers, or paths
- Signature: similar to interface or class types
- Relation between signature and structure in ML is many-to-many

# Structure

```
structure <identifier> =
      struct <elements of the structure> end
```

- Among the structure elements we can find:
  - function definitions
  - exceptions
  - constants
  - types
  - …

# Example

```
structure IntLT = struct
    type t = int
    val lt = (op <)
    val eq = (op =)
end;
```

- Output

```
structure IntLT:
  sig val eq: ''a * ''a -> bool
      val lt: int * int -> bool
      eqtype t
  end
```

# Another definition

- We could also write

```
structure IntDiv = struct
    type t = int
    fun lt (m, n) = (n mod m = 0)
    val eq = (op =)
end;
```

- With the same types (but different interpretations)

```
structure IntDiv:
    sig val eq: ''a * ''a -> bool
        val lt: int * int -> bool
        eqtype t
end;
```

# Long identifiers

- Referring to functions

```
IntLT.lt;
val it = fn: int * int -> bool
IntDiv.lt;
val it = fn: int * int -> bool
```

- Using functions

```
IntLT.lt (3,4);
val it = true: bool
IntDiv.lt(3,4);
val it = false: bool
```

# Signatures

- Specify the type of the structure
- Example

```
signature ORDERED = sig
    type t
    val lt : t * t -> bool
    val eq : t * t -> bool
end;
```

# Queues

```
signature QUEUE =
sig
    type 'a queue
    exception QueueError
    val empty : 'a queue
    val isEmpty : 'a queue -> bool
    val singleton : 'a -> 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
end;
```

# Another example

```
signature STACK =
    sig
        val empty: 'a list
        val pop: 'a list -> 'a option
        val push: 'a * 'a list -> 'a list
        eqtype 'a stack
    end;
```

- Recall:

```
    datatype 'a option = NONE | SOME of 'a
```

# Structure

```
structure Stack = struct
    type 'a stack = 'a list
    val empty = []
    val push = op::
    fun pop [] =NONE
    | pop (tos::rest) =SOME tos
end:> STACK;
```

The declaration :> says that

- Stack is an implementation of the STACK signature
- Components not in the signature are not visible outside

# Operation on Stacks

- Push an item

```
> Stack.push (1, Stack.empty);
val it = [1]: int list
```

- Or,

```
> structure S = Stack;
> S.push (1, S.empty);
```

# Exercise L9.4

- Define a signature SET with
  - Parameterized type
  - Value for empty set
  - Operator to test for membership
  - Operator to add an element to a set
  - Operator to remove an element from a set

# Solution exercise L9.4

```
signature SET =
sig
      type ‘a set

      val emptyset: ‘a set
      val isin: “a -> “a set -> bool
      val addin: “a -> “a set -> “a set
      val removefrom: “a -> “a set -> “a set
end;
```

# Exercise L9.5

- With the signature

```
signature SET =
sig
      type 'a set
end;
```

Add a definition for the structure

# Solution exercise L9.5

```
structure Set =
struct
      type ‘a set = ‘a list;
end :> SET;
```

# Exercise L9.6

- With the signature

```
signature SET =
sig
	type ’a set

	val emptyset: ‘a set
end;
```

Add a definition for the structure and test it

# Solution exercise L9.6

```
structure Set =
struct
      type ‘a set = ‘a list;

      val emptyset = [];
end :> SET;
```

- Test

```
val a = Set.emptyset:
```

# Exercise L9.7

- With the signature

```
signature SET =
sig
        type ‘a set

        val emptyset: ‘a set
        val isin: “a -> “a set -> bool
end;
```

Add a definition for the structure and test it

# Solution exercise L9.7

```
structure Set =
struct
        type ‘a set = ‘a list;

        val emptyset = [];
        fun isin _ []=false
           |isin x (y::ys) = (x=y) orelse isin x ys;
end :> SET;
```

- Test

```
val a = Set.emptyset;
val b = Set.isin 1 a;
```

# Exercise L9.8

- With the signature

```
signature SET =
sig
       type ‘a set

       val emptyset: ‘a set
       val isin: “a -> “a set -> bool
       val addin: “a -> “a set -> “a set
end;
```

Add a definition for the structure and test it

# Solution exercise L9.8

```
structure Set =
struct
      type ‘a set = ‘a list;

      val emptyset = [];
      fun isin _ []=false
         |isin x (y::ys) = (x=y) orelse isin x ys;
      fun addin x L = if (isin x L) then L else x::L;
end :> SET;
```

- Test

```
val a = Set.emptyset;
val b = Set.isin 1 a;
val c = Set.addin 1 a;
val d = Set.isin 1 c;
```

# Exercise L9.9

- With the signature

```
signature SET =
sig
       type ‘a set

       val emptyset: ‘a set
       val isin: “a -> “a set -> bool
       val addin: “a -> “a set -> “a set
       val removefrom: “a -> “a set -> “a set
end;
```

Add a definition for the structure and test it

# Solution exercise L9.9

```
structure Set =
struct
        type 'a set = 'a list;

        val emptyset = [];
        fun isin _ []=false
           |isin x (y::ys) = (x=y) orelse isin x ys;
        fun addin x L = if (isin x L) then L else x::L;
        fun removefrom _ [] = []
            |removefrom x (y::ys) = if (x=y) then ys
                                    else y::(removefrom(x,ys));
end :> SET;
```

- Test

```
val a = Set.emptyset;
val b = Set.isin 1 a;
val c = Set.addin 1 a;
val d = Set.isin 1 c;
val e = Set.removefrom 1 c;
val f = Set.isin 1 e;
```

# Exercise L9.10

- Given the following type for trees:

```
datatype 'a T = Lf | Br of 'a * 'a T * 'a T
```

Define a signature with the following operations

  - Count the number of nodes in a tree
  - Find the depth of a tree
  - Find the mirror image of a tree

# Solution exercise L9.10

```
signature TREE =
        sig
        datatype ’a T = Lf | Br of ’a * ’a T * ’a T
        val count : ‘a T -> int
        val depth :’a T -> int
        val mirror : ‘a T -> ‘a T
end;
```

# Exercise L9.11

- Define a structure for this signature

```
signature TREE =
       sig
       datatype ’a T = Lf | Br of ’a * ’a T * ’a T
       val count : ‘a T -> int
       val depth :’a T -> int
       val mirror : ‘a T -> ‘a T
end;
```

# Solution exercise L9.11

```
structure Tree =
struct
    datatype ’a T = Lf | Br of ’a * ’a T * ‘a T
    fun count Lf = 0
         |count (Br(a,b,c)) = 1+count(b)+count(c);
    fun depth Lf = 0
         |depth (Br(a,b,c)) = if depth1(b)>depth(c) then 1+depth(b)
                                 else  1 + depth(c);
    fun mirror Lf = Lf
        | mirror (Br(v,t1,t2)) = Br(v,mirror(t2),mirror(t1));
end :> TREE;
```

# Summary

- Recap
- User-defined types
- Signatures and structures

# Next time


- Scala