λ

Abstract data types

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

Today

- Recursively defined datatypes in ML
- Data abstraction
- Structures and signatures in ML
- Binary Search Trees in ML
- Trees in ML
- Structure restrictions in ML
- Functors

Next lectures

- No class on Thursday May 1st
- Extra-slot for tutoring on Friday May 11th (11:30 – 12:30)
- ML Challenge on Thursday May 15th
- One of the last classes, we will have the exam simulation

When you have time

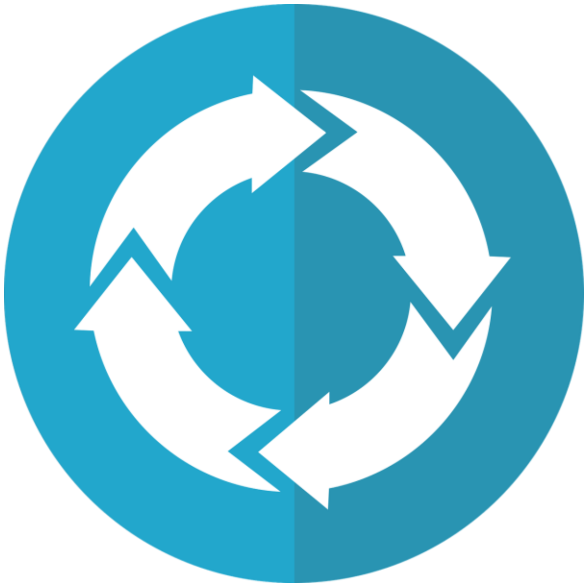
You can find the link also in Moodle!

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
DQDQRX



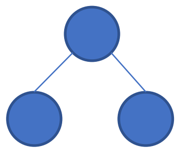
Recursively defined datatypes in ML

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



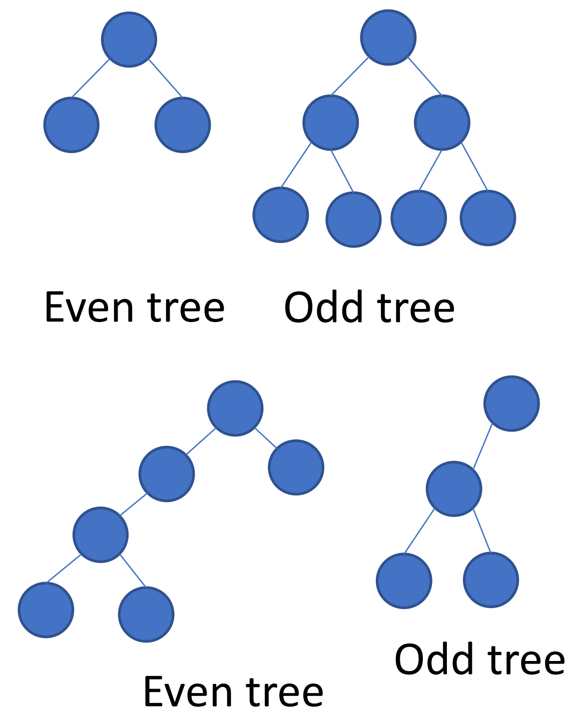
Example of data

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

val myTree =
        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: each path from the root to a node with one or two empty subtrees has an odd number of nodes

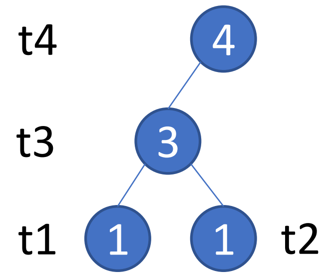


- Simple way to define it:
  - Basis: the empty tree is an even tree
  - Induction: a node with a label and two subtrees that are odd trees is the root of an even tree

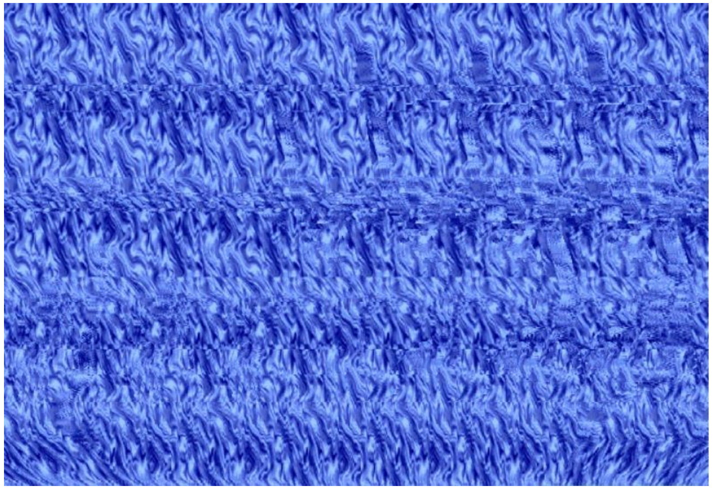
Example

```
> 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
```



Data abstraction

Defining new data types

- When defining new data types, a user can only use existing capsules and a new type does not allow the user to define types at the same level of abstraction of the predefined types
  - It is possible to define new values
  - But the internal structure and operations are still accessible to the programmer

An example

```
type Int_Stack = struct{
    int P[100]; // the stack proper
    int top; // first readable element
}
Int_Stack create_stack(){
    Int_Stack s = new Int_Stack();
    s.top = 0;
    return s;
}
Int_Stack push(Int_Stack s, int k){
    if (s.top == 100) error;
    s.P[s.top] = k;
    s.top = s.top + 1;
    return s;
}
int top(Int_Stack s){
    return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
    if (s.top == 0) error;
    s.top = s.top - 1;
    return s;
}
bool empty(Int_Stack s){
    return (s.top == 0);
}
```

Even in case of equivalence by name, we can access the stack in its representation as an array

```
int second_from_top()(Int_Stack c){
    return c.P[s.top - 1];
    }
```

We would need … linguistic support for abstraction

- Abstraction of control
  - Hide the implementation of procedure bodies
- Data abstraction
  - Hide decisions about the representation of the data structures and the implementation of the operations
  - Example: a stack implemented via
    - A vector
    - A linked list

Abstract Data Types

- One of the major contributions of the 1970s
- Basic idea: separate the interface from the implementation
  - Interface: types and operations that are accessible to the user
  - Implementation: internal data structures and operations acting on the data types
  - Example
    - Sets have operations as `empty`, `union`, `insert`, `is_member?`
    - Sets can be implemented as vectors, lists etc.

Abstract Data Types characteristics

1. A name for the type
2. An implementation or representation for the type (concrete type)
3. Names denoting the operations for manipulating the values of the type with their types
4. For every operation, an implementation that uses the concrete type representation
5. A security capsule which separates the name of the type and those of the operations from their implementations

Concrete languages

- Different languages have different levels of support for ADT
- C:
  - Header file (.h) containing the interface/signature
  - Implementation in separate .c files
- Java, C++:
  - Object-orientation – through classes
    - Methods implementing the interface are public
    - Internal representation private
- ML:
  - Structures and signatures

Structures and signatures in ML

Structures and signatures

- 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
- This is the same mechanism that we have seen for String, Int, Real, … these are all structures

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;
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;
```

It says that ‘a stack is an equality type, that is a type that supports the equality

- 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
```

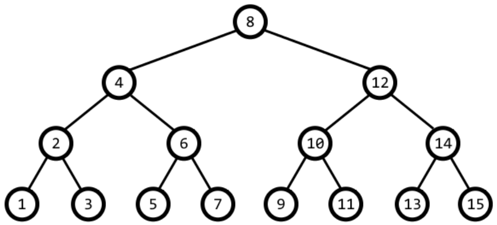
- Or,

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

Components not in the signatures are not visible outside

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

> Stack.hasTop(Stack.empty);
poly: : error: Value or constructor (hasTop) has not been
declared in structure Stack
Found near Stack.hasTop (Stack.empty)
Static Errors
```



Binary Search Trees (BST) in ML

Binary search trees (BST)

- Let us recall

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

- We assume an order predicate `lt(x,y)` that is
  - Transitive: `lt(x,y)`, `lt(y,z)` then `lt(x,z)`
  - Total: either `lt(x,y)` or `lt(y,x)`
  - Irreflective: if `lt(x,y) then not lt(y,x)` and viceversa

- BST property for binary labeled trees: if x is the label of a node n, then for every label y in the left subtree of n, `lt(y,x)` holds, and for every label y in the right subtree of n, `lt(x,y)` holds

How to define `lt`? Examples of order relations

```
> fun intLT (x,y) = x < y;
val intLT = fn: int * int -> bool

> fun lower (nil) = nil
   | lower (c::cs) = (Char.toLower c)::lower (cs);
val lower = fn: char list -> char list

> fun strLT (x,y) =
  implode (lower (explode x)) < implode (lower (explode
y));
val strLT = fn: string * string -> bool
```

Lookup in a BST

```
datatype 'a btree = Empty | Node of 'a * 'a btree * 'a
btree
```

- Write a function `lookup` that given a function `lt`, a BST and an element returns `true` if the element occurs in the binary search tree, `false` otherwise

```
> fun lookup lt Empty x = false
    | lookup lt (Node(y,left,right)) x =
        if lt(x,y) then lookup lt left x
        else if lt(y,x) then lookup lt right x
             else true;
val lookup = fn: ('a * 'a -> bool) -> 'a btree -> 'a ->
bool
```

Example

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

> lookup strLT t "function";
val it = false: bool

> lookup strLT t "ML";
val it = true: bool
```

Insertion into BST

- Write a function insert that given the function `lt` that defines the relation on the BST, a BST and an element e, insert the element in the tree
- The function does not insert into an existing tree but creates a new tree, with the new element added
- A recursive insert that, at each step, creates the appropriate subtree

Insertion

```
> fun insert lt Empty x = Node(x,Empty,Empty)
    |insert lt (T as Node (y,left,right)) x =
        if lt (x,y) then Node (y,(insert lt left x),right)
        else if lt (y,x) then Node (y,left,(insert lt right x))
            else T;
val insert = fn: ('a * 'a -> bool) -> 'a btree -> 'a -> 'a
btree

> insert srtLT t "function";
val it = ("ML",          Node ("as", Node ("a", Empty, Empty),
Node ("in", Node ("function", Empty, Empty), Empty)), Node
("types", Empty, Empty)): string btree
```

Deletion

- Write a function delete that, given a function `lt` over the BST, a BST and the element to delete, deletes the node from the tree
- Also in this case, we return a modified version of the tree. This time, most of the work is in the case of equality
- We first define an auxiliary function `deletemin` which, given a BST, (i) finds the smallest element `y` in the BST `T`, and (ii) the tree that results after deleting this element

- Comments
  - The input to `deletemin` must be a nonempty tree
  - The smallest item will always be the left-most node, so the order relation is not needed

deletemin

```
> exception EmptyTree;
exception EmptyTree

> fun deletemin (Empty) = raise EmptyTree
        | deletemin (Node(y,Empty,right)) = (y,right)
        | deletemin (Node(w,left,right)) =
              let val (y,L) = deletemin(left)
              in (y,Node(w,L,right))
              end;
val deletemin = fn: 'a btree -> 'a * 'a btree
```

Deleting from a tree

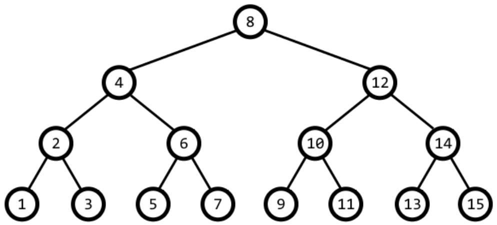
```
> fun delete lt Empty x = Empty
 |delete lt (Node(y,left,right)) x =
        if lt (x,y) then Node(y,(delete lt left x),right)
        else if lt (y,x) then Node(y,left,(delete lt right x))
             else
                case (left,right) of
                    (Empty,r) => r |
                    (l,Empty) => l |
                    (l,r) =>
                    let val (z,r1) = deletemin(r)
                    in Node (z,l,r1)
                    end;
val delete = fn: ('a * 'a -> bool) -> 'a btree -> 'a -> 'a
btree
```

Visiting all the nodes of a tree

- Example: Write a function that sums all the values of a tree

Preorder traversal

- List the label of the root
- In order from the left, list the labels of each subtree in preorder (root, followed by labels in the left tree and then the ones in the right tree)



Trees in ML

Trees

- Datatype tree for general rooted trees (not just binary)

```
> datatype ('label) tree =
          Node of 'label * 'label tree list;
datatype 'a tree = Node of 'a * 'a tree list
```

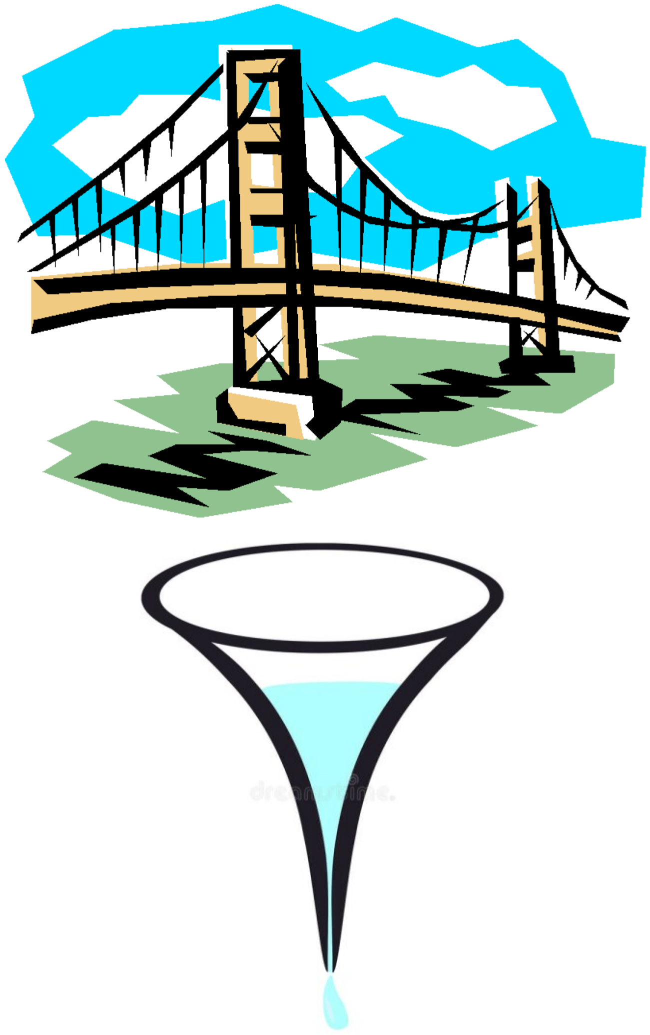
Example

```
> Node (1, [
     Node (2,nil),
     Node (3, [
         Node (4,nil),
         Node (5, [
            Node (7,nil)
         ]),
         Node (6,nil)
     ])
   ]);

val it =
   Node
    (1,
     [Node (2, []),
      Node (3, [Node (4, []), Node (5, [Node (7, [...])]), Node (6, [])])]):
   int tree
```

Example: summing the labels of the nodes

- Write a function `sum` that, given a tree, sums the labels of the nodes



Restricting structures in ML

Example: Mapping structure

```
> structure Mapping = struct
   exception NotFound;
   val create = nil;
   fun lookup (d,nil) = raise NotFound
     | lookup (d,(e,r)::es) =
         if d=e then r
         else lookup (d,es);
    fun insert (d,r,nil) = [(d,r)]
       | insert (d,r,(e,s)::es) =
           if d=e then (d,r)::es
           else (e,s)::insert(d,r,es)
end;

structure Mapping:
  sig
     exception NotFound
     val create: 'a list
     val insert: ''a * 'b * (''a * 'b) list -> (''a * 'b) list
     val lookup: ''a * (''a * 'b) list -> 'b
end
```

A structure that manipulates a list of key-value pairs (d,r). It has a create variable containing an empty list and allows for:

- Searching for the key and returning the value – if in the list
- Inserting a new pair

This is a polymorphic structure.

```
> Mapping.create;
val it = []: 'a list
```

Restricting types through their signatures

- Restrict mappings to be on string int pairs

```
signature SIMAPPING = sig
  val create : (string * int) list;
  val insert : string * int * (string * int) list -> (string * int) list;
  val lookup : string * (string * int) list -> int
end;

signature SIMAPPING =
  sig
   val create: (string * int) list
   val insert: string * int * (string * int) list -> (string * int) list
   val lookup: string * (string * int) list -> int
End

> structure SiMapping : SIMAPPING = Mapping;
structure SiMapping: SIMAPPING
```

Accessing names defined in structures

```
> val m = SiMapping.create;
val m = []: (string * int) list

> val m = SiMapping.insert ("in",6,m);
val m = [("in", 6)]: (string * int) list

> val m = SiMapping.insert ("a",1,m);
val m = [("in", 6), ("a", 1)]: (string * int) list

> SiMapping.lookup ("in",m);
val it = 6: int
```

Opening structures

- Avoid repeating the name of the structure

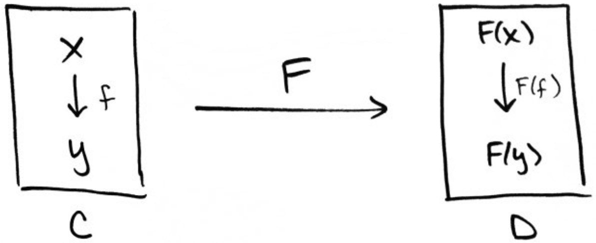
```
> open SiMapping;
val create = []: (string * int) list
val insert = fn: string * int * (string * int) list -> (string *
int)
list
val lookup = fn: string * (string * int) list -> int


> create;
val it = []: (string * int) list


> SiMapping.create;
val it = []: (string * int) list
```

Be very careful with – AVOID OPENING STRUCTURES

- if you open a structure and you overwrite functions, you could assume a semantic that is no more the correct one in the same scope



Functors in ML

Structures and functors

- Structures: Collections of type, datatypes, functions, exceptions, etc., that we want to encapsulate
- Signatures: Collections of information describing the types and other specifications for some of the elements of a structure
- Functors: Operations that take as arguments one or more elements such as structures, and produce a structure.

Why do we need functors?

```
> structure BST = struct
   exception EmptyTree;
   datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree;
   fun lookup lt Empty x = false
       | lookup lt (Node(y,left,right)) x =
           if lt(x,y) then lookup lt left x
           else if lt(y,x) then lookup lt right x else true;
   fun insert lt Empty x = Node(x,Empty,Empty)
       |insert lt (T as Node (y,left,right)) x =
           if lt (x,y) then Node (y,(insert lt left x),right)
           else if lt (y,x) then Node (y,left,(insert lt right x)) else T;
   fun deletemin (Empty) = raise EmptyTree
           | deletemin (Node(y,Empty,right)) = (y,right)
           | deletemin (Node(w,left,right)) = let val (y,L) =
deletemin(left)
                                              in (y,Node(w,L,right))
                                        end;
   fun delete lt Empty x = Empty
      |delete lt (Node(y,left,right)) x =
           if lt (x,y) then Node(y,(delete lt left x),right)
           else if lt (y,x) then Node(y,left,(delete lt right x))
                else case (left,right) of (Empty,r) => r
                                 | (l,Empty) => l
                                 | (l,r) => let val (z,r1) = deletemin(r)
                                              in Node (z,l,r1)
                              end;
end;
```

A structure that manipulates BSTs. Given a generic function lt, it allows for :
- Looking for x
- Inserting a node containing x
- Deleting the node containing x

Why do we need functors?

- Let us assume we want to apply the structure to string BSTs
  - We need to define an appropriate `lt`

```
> fun lower (nil) = nil
    | lower (c::cs) = (Char.toLower c)::lower (cs);
val lower = fn: char list -> char list
> fun lt (x,y) =  implode (lower (explode x)) < implode (lower (explode
y));
val lt = fn: string * string -> bool
```

  - We need to rewrite the structure

Why do we need functors?

```
> structure StringBST = struct
   exception EmptyTree;
   datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree;
   fun lookup Empty x = false
       | lookup (Node(y,left,right)) x =
           if lt(x,y) then lookup left x
           else if lt(y,x) then lookup right x else true;
   fun insert Empty x = Node(x,Empty,Empty)
       |insert (T as Node (y,left,right)) x =
           if lt (x,y) then Node (y,(insert left x),right)
           else if lt (y,x)
                then Node (y,left,(insert right x))
                else T;
   fun deletemin (Empty) = raise EmptyTree
           | deletemin (Node(y,Empty,right)) = (y,right)
           | deletemin (Node(w,left,right)) =
             let val (y,L) = deletemin(left)
                      in (y,Node(w,L,right))
             end;
   fun delete Empty x = Empty
      |delete (Node(y,left,right)) x =
           if lt (x,y) then Node(y,(delete left x),right)
           else if lt (y,x) then Node(y,left,(delete right x))
                else case (left,right) of (Empty,r) => r
                       | (l,Empty) => l
                       | (l,r) => let val (z,r1) = deletemin(r)
                                  in Node (z,l,r1)
                             end;
end;
```

```
structure StringBST:
  sig
    exception EmptyTree
    datatype 'a btree = Empty |
                     Node of 'a * 'a btree * 'a btree
    val delete: string btree -> string -> string btree
    val deletemin: 'a btree -> 'a * 'a btree
    val insert: string btree -> string -> string btree
    val lookup: string btree -> string -> bool
  end
```

Why do we need functors? - Issues

- We have strange types of the elements in the structure
- We do not create an object which is a BST that work on any type with an `lt` (less-then) function

Functors

- Takes a structure and returns another structure
  - As a function takes a value and returns a new value a functor takes a structure and returns a new structure
- Consider our example of BSTs with comparison operator `lt`
- A functor takes as arguments a structure and a `less-than` operator and produces a structure incorporating the comparison operator

```
functor <identifier> (<structure name>:
<signature>) = <structure definition>
```

The steps we take

- Step 1: define a signature `TOTALORDER` that is satisfied by our functor inputs
- Step 2: define a functor `MakeBST` that takes a structure `S` with signature `TOTALORDER` and produces a structure
- Step 3: define structure `STRING` with signature `TOTALORDER` and with a comparison operator on strings
- Step 4: apply `MakeBST` to String to produce the desired structure

Step 1: define the signature `TOTALORDER`

```
> signature TOTALORDER = sig
  type element;
  val lt : element * element -> bool
end;

signature TOTALORDER = sig type element val lt: element *
element -> bool end
```

Step 2: define the functor (sketch)

```
> functor MakeBST (Lt: TOTALORDER):
  sig
    type 'label btree
    exception EmptyTeee;
    val create : Lt.element btree;
    val lookup : Lt.element * Lt.element btree -> bool;
    val insert : Lt.element * Lt.element btree -> Lt.element btree;
    val deletemin : Lt.element btree -> Lt.element Lt.element btree;
     val delete : Lt.element * LT.element btree -> Lt.element btree
  end
=
  struct
   open Lt;
   datatype 'label btree =
           Empty |
           Node of 'label * 'label btree * 'label btree;
    val create = Empty;
    val lookup (x, Empty) = ...
    val insert (x, Empty) = ...
    exception EmptyTree;
    fun deletemin (Empty) = ...
    fun delete (x,Empty) = ...
  end;
```

Step 3: define the functor argument

```
structure String: TOTALORDER =
struct
   type element = string;
   fun lt (x,y) =
     let
        fun lower (nil) = nil
        | lower (c::cs) = (Char.toLower c)::lower (cs);
     in
        implode (lower (explode (x))) < implode (lower( explode
(y)))
      end;
end;
```

Step 4: Apply the functor

```
structure StringBST = MakeBST (String);
```

Extensions: applying functor to explicit structure

```
structure StringBST = MakeBST (
  struct
    type element = string;
    fun lt (x,y) =
       let
         fun lower (nil) = nil
         | lower (c::cs) = (Char.toLower c)::lower (cs);
       in
         implode (lower (explode (x))) < implode (lower(
explode (y)));
       end;
   end
);
```

Summary

- Recursively defined datatypes in ML
- Data abstraction
- Structures and signatures in ML
- Binary Search Trees in ML
- Trees in ML
- Structure restrictions in ML
- Functors

Readings

- Chapter 9 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Intro to lambda calculus