λ

ML Lab 9

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

When you have time

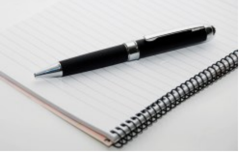
- Fill the feedback form about the course:
  - https://forms.gle/9btC4JBDAWumxeQ29

- The link is also available in Moodle

Next lectures

- Tuesday May 27: short seminar
- Thursday May 29: exam simulation
- Last lecture: Tuesday June 3

Old exercises

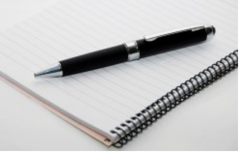


Exercise 7.8

- Define a type `mapTree` that is a specialization of `btree` so that it has a label type that is a set of domain-range pairs
- Define a tree `t1` that has a single node with the pair ("a",1) at the root

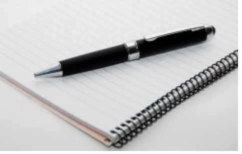
Solution 7.8

```
> type ('d, 'r) mapTree = ('d * 'r) btree;
type ('a, 'b) mapTree = ('a * 'b) btree
> val t1 = Node(("a",1), Empty, Empty): (string, int) mapTree;
val t1 = Node (("a", 1), Empty, Empty): (string, int) mapTree
```



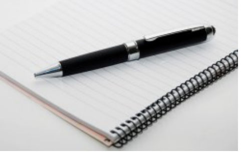
Exercise 7.9

- Write a function `sumTree` for a mapTree `T` of type `(‘a,’b) mapTree` (where the order is defined by the first component). The function visits the tree and returns the sum of the second component of the label of all nodes.
- For instance
  - `sumTree (Node(("a",1), Node(("c",2), Empty, Node(("d",3), Empty, Empty)), Empty) = 6`



Solution 7.9

```
> fun sumTree Empty = 0
    | sumTree (Node((a,b),left,right)) = b + sumTree (left) +
sumTree (right);
val sumTree = fn: ('a * int) btree -> int

> val t2 = Node(("a",1), Node(("c",2), Empty, Node(("d",3),
Empty, Empty)), Empty): (string, int) mapTree;
val t2 = Node    (("a", 1), Node (("c", 2), Empty, Node (("d",
3), Empty, Empty)), Empty): (string, int) mapTree

> sumTree t2;
val it = 6: int
```

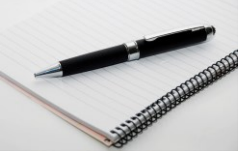


Exercise 8.9

- Given the type `(‘a,’b) mapTree` defined as a particular binary search tree in Exercise 7.8, such that the order is defined by the first component of the tuple:

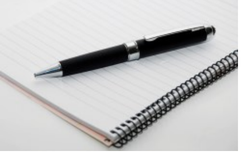
  `type (‘a, ‘b) mapTree = (‘a * ‘b) btree;`

- write a function `lookup lt T` a that searches in tree `T` for a pair `(a, b)`, and, if it finds a pair `(a, b)`, whose first component is `a`, it returns `b`
- The function `lt` should compare domain elements
- If there is no such a pair, return exception `Missing`



Solution 8.9

```
> exception Missing;
exception Missing

> fun lookup lt Empty a = raise Missing
    | lookup lt (Node((c,b),left,right)) a =
        if lt(a,c) then lookup lt left a
        else if lt(c,a) then lookup lt right a
                 else b;
val lookup = fn: ('a * 'a -> bool) -> ('a * 'b) btree -> 'a ->
'b
```

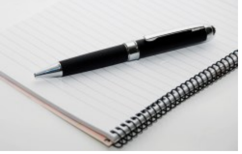


Exercise L8.10

- Given the type `(‘a,’b) mapTree`, write a function assign `lt T a b` that looks in `mapTree T` for a pair `(a, c)`, and, if found, replaces `c` by `b`
- If no such pair is found, `assign` inserts the pair `(a, b)` in the appropriate place in the tree

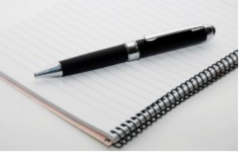
Solution 8.10

```
> fun assign lt Empty a b = Node((a, b), Empty, Empty)
        | assign lt (Node((k, v), L, R)) a b =
               if lt(a, k)
               then Node((k, v), assign lt L a b, R)
               else if lt(k, a)
                       then Node((k, v), L, assign lt R a b)
                       else Node((k, b), L, R);
val assign = fn:
('a * 'a -> bool) -> ('a * 'b) btree -> 'a -> 'b -> ('a * 'b)
btree
```



Exercise 8.1

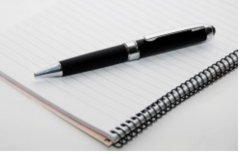
- Define a signature SET with
  - Parameterized type ‘a `set`
  - Value for empty set (`emptyset`)
  - Operator to test the membership of an element to a set (`isin`)
  - Operator to add an element to a set (`addin`)
  - Operator to remove an element from a set (`removefrom`)



Solution 8.1

```
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;
```

Note that here type is actually ‘’a because we have to check for equality

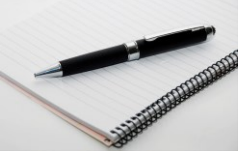


Exercise 8.6

- 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

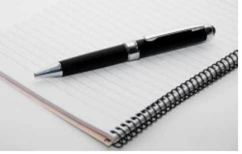


Solution 8.6

```
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;
```



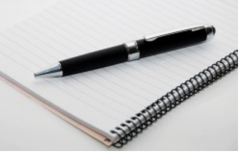
Exercise 8.7

- Given the following type for trees:

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

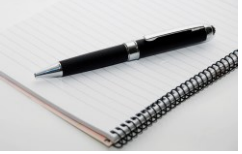
Define a signature `TREE` with the following operations besides the datatype `'a T = Lf | Br of 'a * 'a T * 'a T`

- Count the number of nodes in a tree (`countNodes`)
- Find the depth of a tree (`depth`)
- Find the mirror image of a tree (`mirror`). The mirror image of a tree is a tree in which the right and left subtrees are swapped, e.g.,
  - `mirror Br(3, Br(2,Lf,Lf), Br(5,Br(4,Lf,Lf),Lf) = Br (3, Br(4, Lf, Br(4,Lf,Lf)),Br(2,Lf,Lf))`



Solution 8.7

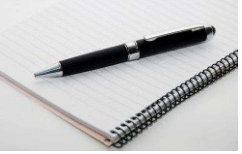
```
signature TREE =
        sig
        datatype ’a T = Lf | Br of ’a * ’a T * ’a T
        val countNodes : ‘a T -> int
        val depth :’a T -> int
        val mirror : ‘a T -> ‘a T
end;
```



Exercise 8.8

- Define a structure `Tree` for this signature

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



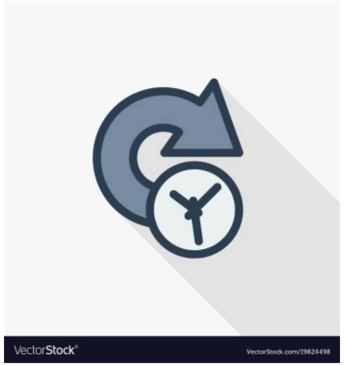
Solution 8.8

```
structure Tree =
struct
    datatype ’a T = Lf | Br of ’a * ’a T * ‘a T
    fun countNodes Lf = 0
         |countNodes (Br(a,b,c)) = 1+countNodes(b)+countNodes(c);
    fun depth Lf = 0
         |depth (Br(a,b,c)) = if depth(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;
```

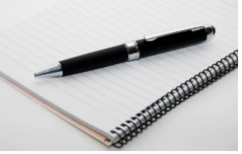
Solution 8.8

In order to access the structure components you have to use the name of the structure. As an alternative you can type `open Tree`, however be careful opening structures – especially the predefined ones, as default functions could be overwritten.

```
> val myTree = Tree.Br(2, Tree.Br(3, Tree.Br(4, Tree.Lf, Tree.Lf),
Tree.Br(5,Tree.Lf,Tree.Lf)),Tree.Br(6, Tree.Lf, Tree.Br(7,Tree.Lf,Tree.Lf)));
val myTree =
   Br (2, Br (3, Br (4, Lf, Lf), Br (5, Lf, Lf)), Br (6, Lf, Br (7, Lf,
Lf))):
   int Tree.T
> Tree.countNodes(myTree);
val it = 6: int
> Tree.depth(myTree);
val it = 3: int
> Tree.mirror(myTree);
val it =
   Br (2, Br (6, Br (7, Lf, Lf), Lf), Br (3, Br (5, Lf, Lf), Br (4, Lf,
Lf))): int Tree.T
```



New exercises

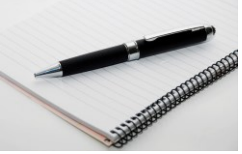


Exercise 9.1

- Given the type `tree` we introduced for representing a general tree `T`

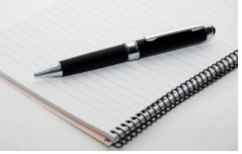
```
datatype ('label) tree =
        Node of 'label * 'label tree list;
datatype 'a tree = Node of 'a * 'a tree list
```

- Write a function `isOn` that given a general tree `T` and an element `x` tells whether `x` appears as a label of `T`
- For instance
  - `isOn (Node(2, [Node (3,nil), Node(5,nil)])) 3 = true`
- You can write the function using recursion or using predefined higher-order functions



Solution 9.1

```
> fun isOn (Node(a,nil)) x = (a=x)
  | isOn (Node(a,t::ts)) x = (a=x) orelse
        isOn t x orelse isOn (Node(a,ts)) x;
val isOn = fn: ''a tree -> ''a -> bool

> fun isOn x (Node(a,L)) =
        (a=x) orelse
            foldr (fn (x,y) => x orelse y) false
                 (map (isOn x) L);
val isOn = fn: ''a -> ''a tree -> bool
```

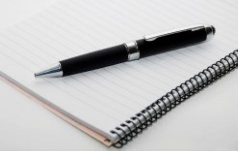


Exercise 9.2

- Given the type `tree` we introduced for representing a general tree `T`

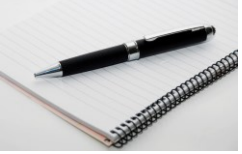
```
datatype (’label) tree =
      Node of ’label * ’label tree list;
datatype ’a tree = Node of ’a * ’a tree list
```

- Write a function `count` that given a general tree `T` and `x`, returns the number of times that `x` appears as a label in `T`
- For instance
  - `count (Node(2, [Node (3,nil), Node(2,nil)])) 2 = 2`
- You can write the function using recursion or using predefined higher-order functions



Solution 9.2

```
> fun countR(Node(a, nil), x) = if a=x then 1 else 0
  | countR(Node(a, t::ts), x) = countR(t, x) + countR(Node(a,
ts), x);
val countR = fn: ''a tree * ''a -> int

>  fun countH(Node(a, L), x) = (if a=x then 1 else 0) +
               foldr (op +) 0 (map (fn t => countH(t, x))L);
val countH = fn: ''a tree * ''a -> int
```

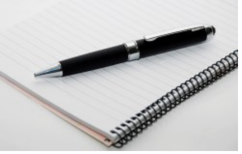


Exercise 9.3

- Given the type `tree` we introduced for representing a general tree `T`

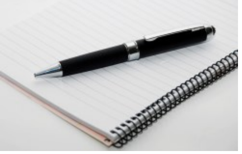
```
datatype ('label) tree =
         Node of 'label * 'label tree list;
datatype 'a tree = Node of 'a * 'a tree list
```

- Write a function `depth` that takes a general tree `T` and returns the depth of `T`
- For instance
  - `depth (Node(2, [Node (3, [Node(4,nil)]), Node(2,nil)]))  = 3`
- HINT: You could need mutually recursive functions
- You can write the function using recursion or using predefined higher-order functions



Solution 9.3

```
> fun max(a, b) = if a<b then b else a;
  fun depthR1(nil) = 1
  | depthR1(t::ts) = max(depthR(t), depthR1(ts))
    and depthR(Node(_, nil)) = 1
  | depthR(Node(a, L)) = 1 + depthR1(L);
  val depthR = fn: 'a tree -> int
  val depthR1 = fn: 'a tree list -> int

> fun depthH(Node(_, L)) = 1 + foldr max 0 (map depthH L);
val depthH = fn: 'a tree -> int
```



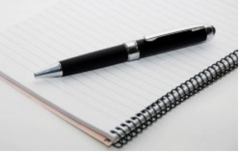
Exercise 9.4

- Given the type `tree` we introduced for representing a general tree `T`

```
datatype (’label) tree =
      Node of ’label * ’label tree list;
```

```
datatype ’a tree = Node of ’a * ’a tree list
```

- Write a function `preOrder` that given a tree `T` returns the list of the nodes of `T` in preorder
- For instance
  - `preOrder (Node(2, [Node (3, [Node(4,nil)]), Node(2,nil)]))  = [2,3,4,2]`
- You can write the function using recursion or using predefined higher-order functions



Solution 9.4

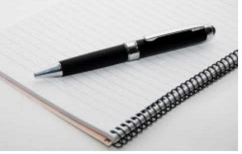
```
> fun preOrder(Node(a,nil)) = [a]
   | preOrder(Node(a,t::ts)) =
          [a] @ preOrder(t) @ (tl(preOrder(Node(a,ts))));
val preOrder = fn: 'a tree -> 'a list


> fun listTreeH(Node(a, L)) = [a] @ foldr (op @) [] (map
listTreeH L);
val listTreeH = fn: 'a tree -> 'a list
```

Exercise 9.5

- Define a structure `Tree` with datatype `tree` representing general trees

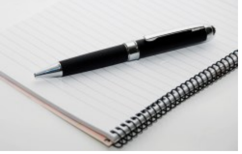
```
datatype ('label) tree =
       Node of 'label * 'label tree list;
datatype 'a tree = Node of 'a * 'a tree list
```

- The structure should contain
  - `create(a)` that returns a node tree with label a
  - `build(a,L)` that returns a tree with root labeled a and list of subtrees `L`
  - `subtree(i,T)` that finds the i-th subtree of the root, with exception `Missing` if it doesn't exist
- For instance
  - `val t = Tree.subtree (2, Node(2, [Node (3, [Node(4,nil)]), Node(2,nil)]): ‘a tree) = Node(2,nil)`



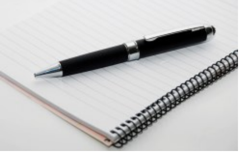
Solution 9.5

```
> structure Tree = struct
    exception Missing;
    datatype ’label tree =
        Node of ’label * ’label tree list;

    (* create a one-node tree *)
    fun create(a) = Node(a,nil);

    (* build a tree from a label and a list of trees *)
    fun build(a,L) = Node(a,L);

    (* find the ith subtree of a tree *)
     fun subtree(i,Node(a,nil)) = raise Missing
        | subtree(1,Node(a,t::ts)) = t
        | subtree(i,Node(a,t::ts)) = subtree(i-1,Node(a,ts));
end;
```



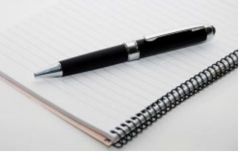
Solution 9.5

```
structure Tree:
  sig
    exception Missing
    val build: 'a * 'a tree list -> 'a tree
    val create: 'a -> 'a tree
    val subtree: int * 'a tree -> 'a tree
    datatype 'a tree = Node of 'a * 'a tree list
end
```



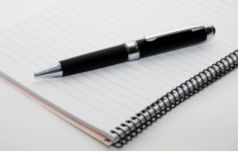
Exercise 9.6

- In the previous exercise, `create(a)` means the same as `build(a,nil)`
- We wish to define a new structure `SimpleTree` that has all the parts of `Tree` except `create`, and restrict the labels to be integers
    - Write a signature `SIMPLE` with these restrictions
    - Use `Tree` and `SIMPLE` to define `SimpleTree`



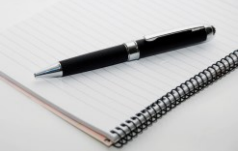
Solution 9.6

```
> signature SIMPLE = sig
        exception Missing;
        datatype 'label tree =
                Node of 'label * 'label tree list;
        val build : int * int tree list -> int tree;
        val subtree : int * int tree -> int tree
 end;
signature SIMPLE =
 sig
        exception Missing
        val build: int * int tree list -> int tree
        val subtree: int * int tree -> int tree
        datatype 'a tree = Node of 'a * 'a tree list
end

> structure SimpleTree: SIMPLE = Tree;
structure SimpleTree: SIMPLE
```



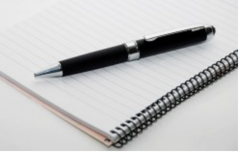
Exercise 9.7

- Use `SimpleTree` to construct a tree with root labeled 1 and three children labeled 2, 3, and 4
- Apply `subtree` to get the second subtree of the root



Solution 9.7

```
> open SimpleTree;
exception Missing
val build = fn: int * int tree list -> int tree
val subtree = fn: int * int tree -> int tree
datatype 'a tree = Node of 'a * 'a tree list

> val t2 = build(2,nil);
val t2 = Node (2, []): int tree
> val t3 = build(3,nil);
val t3 = Node (3, []): int tree
> val t4 = build(4,nil);
val t4 = Node (4, []): int tree
> val t1 = build(1,[t2,t3,t4]);
val t1 = Node (1, [Node (2, []), Node (3, []), Node (4, [])]): int tree
> subtree(2,t1);
val it = Node (3, []): int tree
```

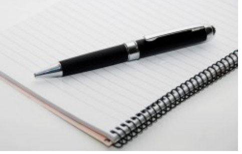


Exercise 9.8

- Define a structure `Stack` representing a stack of elements of arbitrary type (that can be implemented as a list)
- Include the functions
  - `create` to create an empty stack
  - `push x s` that pushes x on top of the stack
  - `pop s` that returns the stack without the top or `EmptyStack` if the stack is empty
  - `isEmpty s` that checks whether the stack is empty
  - `top s` that returns element at the top of the stack or `EmptyStack` if the stack is empty
- Include exception `EmptyStack`

Solution 9.8

```
> structure Stack = struct
 exception EmptyStack;
 type 'a stack = 'a list;

 fun create() = nil: 'a list;
 fun push(x, xs) = x::xs;
 fun pop(nil) = raise EmptyStack
  | pop(x::xs) = xs;
 fun isEmpty(nil) = true
  | isEmpty(S) = false;
 fun top(nil) = raise EmptyStack
  | top(x::xs) = x;
end;
```

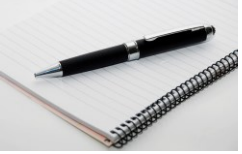


Solution 9.8

```
structure Stack:
  sig
    exception EmptyStack
    val create: unit -> 'a list
    val isEmpty: 'a list -> bool
    val pop: 'a list -> 'a list
    val push: 'a * 'a list -> 'a list
    eqtype 'a stack
    val top: 'a list -> 'a
  end
```

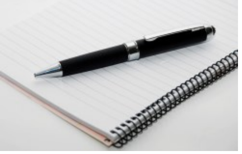
Exercise 9.9

- Design a signature to allow us to create a structure `StringStack` whose elements are of type string, and that omits the `top` operation



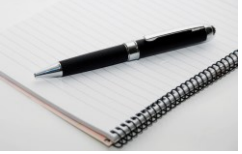
Solution 9.9

```
> signature STRINGSTACK = sig
  type 'a stack;
  val create: unit -> string stack;
  val push: (string * string stack) -> string stack;
  val pop: string stack -> string stack;
  val isEmpty: string stack -> bool;
end;
 sig
    val create: unit -> string stack
    val isEmpty: string stack -> bool
    val pop: string stack -> string stack
    val push: string * string stack -> string stack
    type 'a stack
  end

> structure StringStack: STRINGSTACK = Stack;
structure StringStack: STRINGSTACK
```



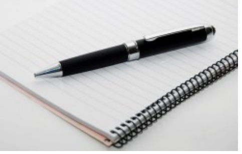
Exercise 9.10

- Design a structure `Queue` that represents a queue of elements (and implements it through a list of elements).
- Operations:
  - `create:` creates an empty queue
  - `enqueue:` adds an element to the end and returns the result
  - `dequeue:` returns the pair of the first element and the rest of the queue
  - `isEmpty`
  - Exception `EmptyQueue`



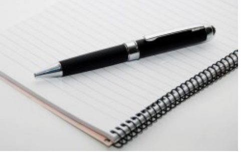
Solution 9.10

```
> structure Queue = struct
  exception EmptyQueue;
  type 'a queue = 'a list;
  val create = [];
  fun enqueue(x,Q) = Q@[x];
  fun dequeue(nil) = raise EmptyQueue
  | dequeue(q::qs) = (q,qs);
  fun isEmpty(nil) = true
  | isEmpty(_) = false;
end;
```



Exercise 9.11

- Design a signature to create a structure `PairQueue` of pairs consisting of a string and an integer with the same operations of `Queue`



Solution 9.11

```
> signature PAIRQUEUE = sig
   type 'a queue;
   val create: (string * int) queue;
   val enqueue: ((string * int) * (string * int) queue) -> (string * int) queue;
   val dequeue: (string * int) queue -> (string * int) * (string * int)queue;
   val isEmpty: (string * int) queue -> bool;
end;
signature PAIRQUEUE =
  sig
    val create: (string * int) queue
    val dequeue:
        (string * int) queue -> (string * int) * (string * int) queue
   val enqueue:
       (string * int) * (string * int) queue -> (string * int) queue
   val isEmpty: (string * int) queue -> bool
   type 'a queue
end

> structure PairQueue: PAIRQUEUE = Queue;
structure PairQueue: PAIRQUEUE
```

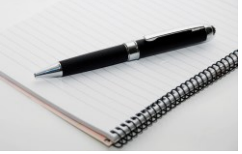
Let’s recall …

- A functor is a structure that 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

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

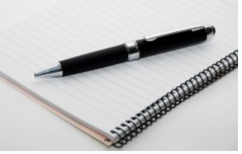
Let’s recall

- The steps we took for building a functor `MakeBST` that allows for building a binary search tree with a customized ordering function
- 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



Exercise 9.12

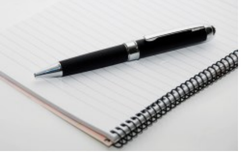
- Write a structure `IntTriple` implementing the signature `TOTALORDER` that, in place of `String`, defines the elements of a binary search tree as tuples of 3 integer numbers
- lt: (a, b, c) < (x, y, z) iff
  - a < x
  - a = x and b < y
  - a = x, b = y and c < z
- Use the functor `MakeBST` to get a structure that stores triples of integers in binary trees



Exercise 9.12

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

functor MakeBST(Lt: TOTALORDER):
  sig
    type 'label btree;
    exception EmptyTree;
    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
=
```

Actually you can omit the signature here, although it is better to specify it

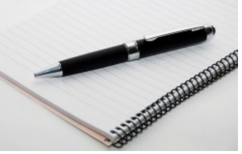


Exercise 9.12

```
struct
  open Lt;
  datatype 'label btree
      = Empty
       | Node of 'label * 'label btree * 'label btree;
   val create = Empty;
   fun lookup(x, Empty) = false
    | lookup(x, Node(y, left, right)) =
       if lt(x, y)
       then lookup(x, left)
       else if lt(y, x)
       then lookup(x, right)
       else (* x=y *) true;
```

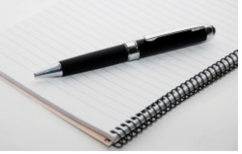
Exercise 9.12

```
fun insert(x, Empty) = Node(x, Empty, Empty)
  | insert(x, T as Node(y, left, right)) =
     if lt(x, y)
     then Node(y, insert(x, left), right)
     else
       if lt(y, x)
       then Node(y, left, insert(x, right))
       else (* x=y *) T; (* do nothing; x was already there *)
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;
```



Exercise 9.12

```
fun delete(x, Empty) = Empty
    | delete(x, Node(y, left, right)) =
      if lt(x, y)
      then Node(y, delete(x, left), right)
      else if lt(y, x)
      then Node(y, left, delete(x, right))
      else (* x=y *) 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;
```



Solution 9.12

```
structure IntTriple: TOTALORDER = struct
   type element = int * int * int;
   fun lt((l1, l2, l3), (r1, r2, r3)) =
   l1<r1 orelse l1=r1 andalso l2<r2 orelse l1=r1 andalso l2=r2 andalso
l3<r3;
end;
structure IntTriple: TOTALORDER
```