Exercise

Define a function `flatten` that takes a list of lists and returns the list consisting of all the elements, in the same order in which they appear in the argument

Example

```
flatten [[1,2],[2,3,4],[5],[],[6,7]] = [1,2,2,3,4,5,6,7]
```

Solution

```
fun flatten [] = []
  | flatten (x::l) = x @ flatten l;


flatten [] ;
flatten [[]];
flatten [["a"],["b","a"]];
```

Exercise

Consider the "binary tree" data structure:

```
datatype 'a btree = emptybt | consbt of 'a * 'a btree * 'a btree;
```

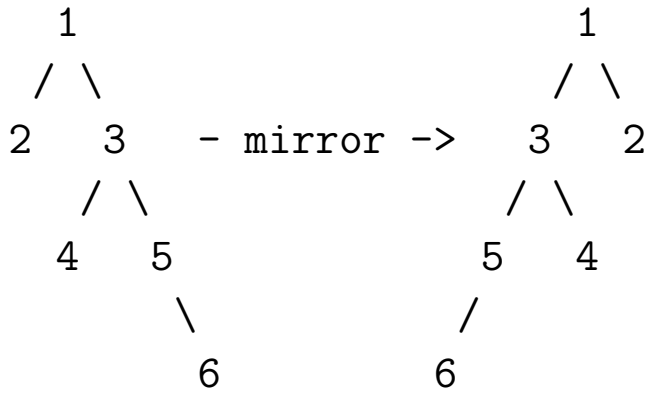
Define a function `sum_tree : int btree -> int` which returns the sum of all the elements of the tree

Solution

```
fun sum_tree emptybt = 0
  | sum_tree (consbt(x,t1,t2)) = x + sum_tree t1 + sum_tree t2;
```

Exercise

Define a function `mirror : 'a btree -> 'a btree` which returns the "mirror" of the input tree.



Solution

```
fun mirror emptybt = emptybt
  | mirror (consbt(x,t1,t2)) = consbt(x, mirror t2, mirror t1);
```

Signatures and structures

Introduction

- Signature: Similar to interface or class types
- Relation between signature and structure in ML is many-to-many
- Structure: sequence of declarations comprising the components of the structure
  - A structure may be bound to a structure variable using a *structure binding*
  - The components of a structure are accessed using *long identifiers*, or *paths*

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

- Refering 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

```
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 peek      : 'a queue -> 'a
    val remove    : 'a queue -> 'a * 'a queue
 end;
```

Implementation

- A structure with this signature

```
structure TwoListQueue    :> QUEUE =
struct
  type 'a queue = 'a list * 'a list
  exception QueueError

  val empty = ([],[])

  fun isEmpty ([],[]) = true
    | isEmpty _ = false

  fun singleton a = ([], [a])
```

Implementation

```
  fun insert (a, ([], [])) = ([], [a])
    | insert (a, (ins, outs)) = (a::ins, outs)

  fun peek (_,[]) = raise QueueError
    | peek (ins, a::outs) = a

  fun remove (_,[]) = raise QueueError
    | remove (ins, [a]) = (a, ([], rev ins))
    | remove (ins, a::outs) = (a, (ins,outs))

end
```

Implementation

- The declaration `:>` says that
  - `TwoListQueue` is an implementation of the `QUEUE` signature
  - Any type components not in the signature are not visible outside

Exercise

- Define a signature `SET` with
  - 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

```
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

With the signature

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

Add a definition for the structure

Solution

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


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

Exercise

With the signature

```
signature SET =
sig
  type 'a set

  val emptyset   : 'a set
end;
```

Add a definition for the structure

Solution

```
signature SET =
sig
  type 'a set
  val emptyset   : 'a set
end;


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

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

Test

```
val a = Set.emptyset;
```

Exercise

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

Solution

```
signature SET =
sig
  type 'a set
  val emptyset   : 'a set
  val isin       : ''a -> ''a set -> bool
end;

structure Set =
struct
  type 'a set = 'a list;
   val emptyset = [];
   val rec isin =  fn x =>  (fn []   => false
          | y::l =>
   if (x = y)  then   true
      else    isin x l);
end :> SET;


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

Exercise

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

Solution

```
structure Set =
struct
  type 'a set = 'a list;
   val emptyset = [];
   val rec isin =     fn x =>      (fn []   => false
        | y::l =>   if (x = y)   then   true   else    isin x l);

  val addin = fn x =>  fn l =>
       if (isin x l)  then l
       else x::l;
end :> SET;

val a = Set.emptyset;
val b = Set.addin 1 a;
val c = Set.isin 1 b;
val d = Set.isin 2 b;
```

Exercise

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

Solution

```
structure Set =
struct
  type 'a set = 'a list;
   val emptyset = [];
   val rec isin =     fn x =>      (fn []   => false
        | y::l =>   if (x = y)   then   true   else    isin x l);

  val addin = fn x =>  fn l =>
       if (isin x l)  then l
       else x::l;
  val rec removefrom =fn x =>(fn [] => []
       | y::l =>
         if (x = y)  then l
      else y::(removefrom x l));
end :> SET;

val a = Set.emptyset;
val b = Set.addin 1 a;
val c = Set.addin 2 b;
val d = Set.isin 1 c;
val e = Set.removefrom 1 c;
val f = Set.isin 1 e;
```

Exercise

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

```
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 reflect : 'a T -> 'a T
end;
```

Exercise

Define a structure with this signature

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

Another example: A Stack

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

Note:

```
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;
```

Operations on Stacks

- Push an item

```
Stack.push (1, Stack.empty);
```

- Or,

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

- But we want `int stack` not `int list`

Stacks

- First attempt

```
S.push (1, S.empty) : int S.stack;
val it = [1] : int S.stack
```

- We convinced the type system to use a different type, but ML then just considers `S.stack` to be an alias for `list`
- For this we need to use signatures

Signatures

```
signature Stack = sig
type 'a stack
val empty : 'a stack
val push :'a * 'a stack->'a stack
val pop : 'a stack -> 'a option
end;
```

Signature instead of aliasing

- Using the stack we defined before

```
structure LS = Stack :> Stack;
```

- Use the stack

```
LS.push (1, LS.empty);
```

- This is the type we wanted

Signatures

- Stack type is now different from list type

```
val l : int list = LS.push (1, LS.empty);
```

- We get a type error