Binding in ML

```
val x=3;
fun addx(a) = a+x;
val x=10.0;
addx(2);
```

Result

```
val it = 5: int
```

Exercise

Suppose we execute

```
val a=2;
fun f(b) = a*b;
val b=3;
fun g(a) = a+b;
```

What are the results of:

- `f(4)`
- `f(4)+b`
- `g(5)`
- `g(5)+a`
- `f(g(6))`
- `g(f(6))`

Solution

- 8
- 11
- 8
- 10
- 18
- 15

Exercises on Lists

Exercise

Test whether a list is ordered

Solution

```
fun isordered l =
   if length l <= 1 then true
   else if car(l) > car(cdr(l )) then false
   else isordered(cdr(l));
```

Exercise

Test if ordered lists are equal

Solution

```
val rec equal =
  fn (l1,l2) =>
    if (isempty l1) then
       if (isempty l2) then  true else false
       else if (car l1 = car l2) then  equal (cdr l1, cdr l2)
       else false;
```

Exercise

Test if two *unordered* lists are equal

Solution

- Test membership of a list

```
val rec ismember =
   fn (a,l) =>
      if isempty (l) then false
      else if (a = car l) then true
      else ismember (a,cdr l);
```

- List containment

```
val rec containedin = fn(l1,l2) =>
   if isempty (l1) then true
   else ismember (car l1,l2) andalso containedin(cdr l1,l2)  ;
```

- Equality

```
val rec equallist = fn(l1,l2) =>
  containedin(l1,l2) andalso containedin (l2,l1);
```

Exercise

Write a function `sum` that sums a list of integers.

Solution

```
val rec sum = fn
     empty => 0
     | cons (a, l)=> a + sum l;
```

Lists in ML

Types

- Tuples

```
val x = (5, 1.4);
```

- Lists

```
val x = [5,6];
```

- But not

```
val x = [5.5,6];
```

The LIST signature

- Basic components of an abstract data type
  - Signature: Operators and their types
  - Structure: Implementation of these operators
- For lists, we shall only study the signature

Signature, Part 1

```
datatype 'a list = nil | :: of 'a * 'a list
exception Empty

val null : 'a list -> bool
val length : 'a list -> int
val @ : 'a list * 'a list -> 'a list
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val last : 'a list -> 'a
val getItem : 'a list -> ('a * 'a list) option
val nth : 'a list * int -> 'a
val take : 'a list * int -> 'a list
val drop : 'a list * int -> 'a list
val rev : 'a list -> 'a list
val concat : 'a list list -> 'a list
val revAppend : 'a list * 'a list -> 'a list
val app : ('a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
```

Signature, Part 2

```
val find : ('a -> bool) -> 'a list -> 'a option
val filter : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool)
                  -> 'a list -> 'a list * 'a list
val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
val exists : ('a -> bool) -> 'a list -> bool
val all : ('a -> bool) -> 'a list -> bool
val tabulate : int * (int -> 'a) -> 'a list
val collate : ('a * 'a -> order)
                -> 'a list * 'a list -> order
```

Lists

- Null

```
val a = [1,2,4,5,3];
val b = [];

null a;
List.null a;
null b;
List.null b;
```

- Note: For some operators, the `List` prefix is optional
- Length

```
length a;
length b;

val c=5;
length c;
```

Lists

- Concatenation (defined as an infix operator)

```
val c = [3,4];
a @ c;
```

- Head and tail (same as car and cdr)

```
a;
hd a;
tl a;
```

- But not

```
hd b;
tl b;
```

Lists

- Last

```
List.last a;
```

Lists

- Get first item

  ```
  List.getItem a;
  ```

- But what if there is no first item?
- Result is of type `('a * 'a list) option`. `option` is a type with two cases:
  - `SOME` followed by the first item and the rest is it exists
  - `NONE`, otherwise

Lists

- Selecting an item

```
List.nth (a,1);
List.nth (a,8);
```

- Take, and drop, the first 3 elements

```
List.take (a,3);
List.drop (a,3);
```

Lists

- Reverse a list

```
rev a;
```

- Argument: A list of lists. Concatenate all of them

```
List.concat [a,c];
List.concat [a,b,c];
```

- Reverse append

```
List.revAppend (a,c);
```

Lists

- Apply function

```
fun square n = n * n;
List.map square a;
```

- `List.app` for functions that do not return a value

Partial Map

- Function that returns an optional value
- `mapPartial` takes only those values that have the form `SOME`

Filter

```
val a = [1,0,~1,2];
fun positive x = x >0;
List.filter positive a;
```

Lists

- Partition

```
val a = [1,~1,2,~2,0];
fun positive n = n >0;
List.partition positive a;
```

Folding

- Summing a list

```
fun sum (l:int list):int =
  case l of
    [] => 0
  | x::xs => x + (sum xs);
```

- Concatenating a list of strings

```
fun concat (l:string list):string =
  case l of
    [] => ""
  | x::xs => x ^ (concat xs);
```

Folding

- Both functions are very similar
- In both cases, we traverse a listm performing some operation with the data at each step
- How can we generalize this?
- As we traverse a list, we use an accumulator
    - For example, if we sum the integers, we could store the current sum in the accumulator
    - We start with the accumulator set to 0, and add each element we reach to the accumulator.
    - At the end, return the value stored in the accumulator.

Folding

- Summing a list, with an accumulator

```
fun sum' (acc:int) (l:int list):int =
  case l of
    [] => acc
  | x::xs => sum' (acc+x) xs;
```

- Concatenating strings, with an accumulator

```
fun concat' (acc:string) (l:string list):string =
  case l of
    [] => acc
  | x::xs => concat' (acc^x) xs;
```

Folding

- With `foldl`

```
fun sum (l:int list):int = foldl (fn (x,acc) => acc+x) 0 l;
fun concat (l:string list):string = foldl (fn (x,acc) => acc^x) "" l;
```

  Notice the inline definition of (anonymous) functions

- If `f` is a function of type `'a*'b->'b`, the expression `foldl f b[x1,x2,...,xn]` evaluates to `f(xn, f(..., f(x2, f(x1,b))))`
- `foldr` traverses the list right to left, evaluating to `f(x1,f (x2,f(...,f(xn,b))))`

Folding

- A more natural definition of concatenation

```
fun concat (l:string list):string = foldr (fn (x,acc) => x^acc) "" l;
```

- Using currying, we can omit the list argument

```
val sum = foldl (fn (x,a) => x+a) 0;
val concat = foldr (fn (x,a) => x^a) "";
```

Infix notation

- Infix binary operators: Can be made into functions by qualifying their name with the structure they belong to
- `Int.+` is therefore a function that takes in two integers and adds them
- We can therefore write

```
val sum = foldl Int.+ 0;
val concat = foldr String.^ "";
```

Folding

- Many list functions can be expressed with folding

```
fun length l = foldl (fn (_,a) => a+1) 0 l;
```

More list operators

- Exists

```
val a = [1,2,3,4];
List.exists (fn a => a<2) a;
List.exists (fn a => a<1) a;
```

- All

```
List.all (fn a => a<5) a;
List.all (fn a => a<2) a;
```