In acroread, it is useful to enable text selection in the hand tool (Edit → Preferences → General), because this allows copy-pasting text from acroread to an editor and subsequently from the editor to the ocaml interpreter. (Code can be copy-pasted directly, types cannot: the ' end up as the wrong character.)

1 Lists

On the pre-defined list type, we have seen the definitions of several functions. We repeat a few of them.

```
let rec map f xs = match xs with
  | [] -> []
  | a::x -> (f a)::(map f x)
;;

let rec fold_left op e xs = match xs with
  | [] -> e
  | (a::x) -> fold_left op (op e a) x
;;

let rec fold_right op xs e = match xs with
  | [] -> e
  | (a::x) -> op a (fold_right op x e)
;;

let rec filter p xs = match xs with
  | [] -> []
  | a::x when p a -> a::(filter p x)
  | a::x -> filter p x
;;

let rec split p xs = match xs with
 | [] -> ([],[])
 | a::x -> let xs,ys=split p x in
             if p a then (a::xs,ys) else (xs,a::ys)
;;

let hd = function
 | Cons(a,_) -> a
 | Nil -> failwith "hd must be called on non-empty list"
;;

let tl = function
 | Cons(_,x) -> x
 | Nil -> failwith "tl must be called on non-empty list"
;;
```

Define the following functions yourself:

take `take` *n list* returns the first *n* elements *list*, or the whole list if the list is shorter than *n* elements.

```
take 2 [1;2;3;4] = [1;2]
take 2 [1] = [1]
```

drop `drop` *n list* returns the list without its first *n* elements:

```
drop 2 [1;2;3;4] = [3;4]
drop 2 [1] = []
```

even `even` *n* returns true iff *n* is even;

odd `odd` *n* returns true iff *n* is odd;

take while `take_while` *p list* returns the largest possible prefix of *list* in which every element satisfies *p*:

```
take_while even [2;4;5;6] = [2;4]
```

drop while `drop_while` *p list* returns the list without the largest possible prefix of *list* in which every element satisfies *p*:

```
drop_while even [2;4;5;6] = [5;6]
```

quicksort `quicksort` *list* returns a sorted list with the same elements as *list*. The algorithm used is: if the list is empty, return it. If the list is nonempty, split the list into two parts: one part containt the elements less than or equal to the first element the other part the elements larger than the first element. Sort both parts and append them to produce the result.

E.g. to sort [2;1;3;4], we split it into [2;1] and [3;4]. If we sort those, we get [1;2] and [3;4], so the result is [1;2;3;4].

mergesort Sort recursively by: splitting lists of length 2 or more into lists of the same length (plus or minus one), sorting those lists and then merging the two sorted lists into a single sorted list.

```
          [4;1] -> [1;4] = [1;   ;4]
         /                          \
[4;3;1;2]                            [1;2;3;4]
         \                          /
          [3;2] -> [2;3] = [  2;3  ]
```

run length encoding Run length encoding is a simple compression method where a list of elements is converted into a list of element,integer pairs where the (e, n) represents e repeated n times. Define encoding and decoding functions such that:

```
rle_encode [3;2;2;1;1;1] = [(3, 1); (2, 2); (1, 3)]
rle_decode [('a',1);('b',2);('c',3)] = ['a'; 'b'; 'b'; 'c'; 'c'; 'c']
```

2 Trees

In the lecture, we defined labeled trees as

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree;;
```

and the following three functions on trees:

```
let rec size = function
 | Leaf -> 0
 | Node(a,t1,t2) -> 1+size(t1)+size(t2)
;;

let rec depth = function
 | Leaf -> 0
 | Node(a,t1,t2) -> 1 + max (depth t1) (depth t2)
;;

let rec map f = function
 | Leaf -> Leaf
 | Node(a,t1,t2) -> Node(f a,map f t1,map f t2)
;;
```

Implement the following functions yourself:

AVL check Implement checking of the depth condition of AVL trees. (The depths of the children of any node differ by at most one.)

```
avl_check(Node(2,Leaf,Leaf))=true
avl_check(Node(2,Leaf,Node(2,Leaf,Leaf)))=true
avl_check(Node(2,Leaf,Node(2,Leaf,Node(2,Leaf,Leaf))))=false
```

flattening Binary trees can be visited in three orders

pre order Visit current node then nodes in left child then nodes in right child.

in order Visit nodes in left child then current node then nodes in right child.

post order Visit nodes in left child then nodes in right child then current node.

Implement the three visiting orders for transforming a tree into a list:

```
pre_order(Node(1,Node(2,Leaf,Leaf),Node(3,Leaf,Leaf)))  = [1;2;3]
in_order(Node(1,Node(2,Leaf,Leaf),Node(3,Leaf,Leaf)))   = [2;1;3]
post_order(Node(1,Node(2,Leaf,Leaf),Node(3,Leaf,Leaf))) = [2;3;1]
```