

1. Gives the types of the following Ocaml functions. (All are type-correct.)

(a) `let a x y = x+y`

(b) `let b (x,y) = x+y`

(c) `let c (x,y) = x`

(d) `let d x = match x with (a,b) -> a`

(e) `let e x = hd x + 1`

(f) `let f x y = match x with  
[] -> 0 | a::b -> a+y`

(g) `let g (a,b) (c,d) = (a+d, b^c)`  
(Recall that `^` is the string concatenation operation.)

(h) `let rec h x = match x with  
[] -> 0 | (a,b)::r -> a + (h r)`

**val a : int -> int -> int**  
**val b : int \* int -> int**  
**val c : 'a \* 'b -> 'a**  
**val d : 'a \* 'b -> 'a**  
**val e : int list -> int**  
**val f : int list -> int -> int**  
**val g : int \* string -> string \* int -> int \* string**  
**val h : (int \* 'a) list -> int**

2. Define the following Ocaml functions:

(a) `genlist m n = [m; m+1; ... ; n]` (or `[]` if `m>n`)

**let rec genlist m n = if m>n then [] else m :: (genlist (m+1) n)**

(b) `concatWithComma [s1; s2; ...; sn] = s1 ^ "," ^ s2 ^ ... ^ sn`

**let rec concatWithComma sl =  
 match sl with  
 [] -> ""  
 | [s] -> s  
 | s::ss -> s ^ "," ^ concatWithComma ss**

(c) `compress: int list -> (int * int) list` replaces runs of the same integer with a pair giving the count and the number. E.g.

`compress [1;1;5;6;6;6;3] = [(2,1); (1,5); (3,6); (1,3)]`

```

let rec compress lis = if lis = [] then [] else
  match compress (tl lis) with
  [] -> [(1, hd lis)]
  | (n,x)::lis' -> if x = hd lis
    then (n+1,x):: lis'
    else (1, hd lis)::(n,x)::lis'

```

(d) apply: string -> int list -> int applies the operator described by the string argument to the elements in the int list. The string argument can be either "times" or "plus".

```

apply "times" [2;3;4] = 24

```

Assume the int list argument is non-empty.

```

let rec apply s lis = match lis with
  [n] -> n
  | n::lis' -> let n' = apply s lis'
    in if s="times" then n*n' else n+n'

```

3. Suppose we are given the following type definition:

```

type btree = leaf of int | node of int * btree * btree

```

Define the following functions in Ocaml:

(a) preorder: btree -> int list gives the preorder listing of the labels of the tree. E.g.

```

let t = node(1, node(2, leaf(4), leaf(5)), node(3, leaf(6), leaf(7)))
preorder t
=> [1; 2; 4; 5; 3; 6; 7]

```

```

let rec preorder bt = match bt with
  Leaf n -> [n]
  | Node(n,lt,rt) -> n :: (preorder lt @ preorder rt)

```

(b) followpath: btree -> boolean list -> int list gives the list of integers in the tree on the path described by the boolean list, where "true" means follow the left child and "false" means follow the right child. You may assume that the path described by the boolean list actually exists in the tree. E.g.

```

followpath t [true; false]
=> [1; 2; 5]

```

```

let rec followpath bt blis = match bt with
  Leaf n -> [n]
  | Node(n,lt,rt) -> if blis = [] then [n]
    else if hd blis
      then n::(followpath lt (tl blis))
      else n::(followpath rt (tl blis))

```

(c) `height: btree -> int` gives the height of a tree, defined as the length of the longest path from the root to a leaf node.

```
height t;;  
=> 2
```

**let max (x,y) = if x>y then x else y**

**let rec height bt = match bt with**

**Leaf n -> 0**

**| Node(n,lt,rt) -> 1 + max(height lt, height rt)**

(d) `balanced: btree -> boolean` returns true if for every internal node, the heights of its children differ by no more than 1.

```
balanced t  
=> true  
let t1 = node(1, leaf(2), node(3, leaf(6), leaf(7)))  
=> true  
let t2 = node(1, leaf(2), node(3, leaf(6), node(8, leaf(7),  
leaf(9)))  
=> false
```

**let rec balanced bt = match bt with**

**Leaf n -> true**

**| Node(n,lt,rt) -> let h1 = height lt**

**and h2 = height rt**

**in let p = if h1<h2 then h2-h1 else h1-h2**

**in (p=0 || p=1) && balanced lt**

**&& balanced rt**

4. Given this Java class definition:

```
class List { public int h; public List t;  
  public int hd () { return h; }  
  public List tl () { return t; }  
  public List (int h, List t) {this.h = h; this.t = t; }  
}
```

The empty list is represented by null. Define the Java function to concatenate two lists:

```
public static append (L1, L2) {  
  
  if (L1 == null)  
    return L2;  
  return new List(L1.h, append(L1.t, L2));  
  
}
```

5. Given the grammar:

```
E -> E + T | T
T -> T * P | P
P -> id | ( E )
```

and an Ocaml type for trees:

```
type tree = Node of string * tree list
```

we can represent concrete syntax trees. For example, the syntax tree



would correspond to the term

```
Node("E", [Node("E", [Node("T", [Node("P", [Node("id", [])])])])]);
      Node("+", []);
      Node("T", [Node("P", [Node("id", [])])])])
```

Here is a type for abstract syntax :

```
type exp = Id of string | Plus of Exp*Exp | Times of Exp*Exp
```

Write a function `abstract:tree -> exp` to transform a concrete syntax tree to an AST.

**let rec abstract t = match t with**

**Node(s, children) ->**

**(match children with**

**[] -> Id s**

**| [ch] -> abstract ch**

**| [Node("(", []); ch; Node(")", [])] -> abstract ch**

**| [ch1; Node(op, []); ch2] ->**

**let ach1 = abstract ch1**

**and ach2 = abstract ch2**

**in if op = "+" then Plus(ach1, ach2)**

**else Times(ach1, ach2) )**