

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)`

2. Define the following Ocaml functions:

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

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

(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)]`

(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.

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]
```

(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]
```

(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
```

(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
```

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) {
}
}
```

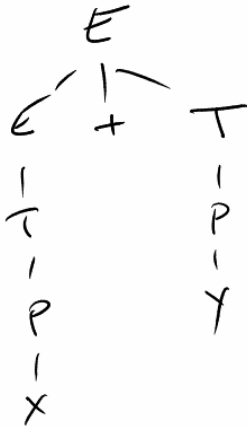
5. Given the grammar:

```
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.