Exercise Week 9 (Michael R. Hansen, 19-11-20)

Q 1) f's type.

From the first part of the declaration

let rec f x = function | [] -> [] | ...

we see that f's type must have the form

f: 'a -> 'b list -> 'c list.

From the second clause

| y::ys -> (x,y) :: f x ys

we have y: 'b , (x,y): 'a * 'b and 'c = 'a * 'b.

Therefore, f: 'a -> 'b list -> ('a * 'b) list

as there are no further constraints.

allPairs' type.

From the first part of the declaration

let rec allPairs xs ys = match xs with
| [] -> []
...

we see that allPairs' type has the form:

allPairs: $\underbrace{\text{'a list}}_{\text{type of xs}}$ -> 'c -> 'd list

From

| x :: xrest -> f x ys @ allPairs ....

we get that ys is a list because f: 'a -> 'b list -> ('a * 'b) list

i.e. ys : 'b list , 'c = 'b list and

'd = 'a * 'b.

Since there is no further constraint:

~~f~~ allPairs: 'a list -> 'b list -> ('a * 'b) list

Q 2)

f "a" [1; 2; 3]
⇝ ("a",1) :: f "a" [2; 3]
⇝ ("a",1) :: ("a",2) :: f "a" [3]
⇝ ("a",1) :: ("a",2) :: ("a",3) :: f "a" []
⇝ ("a",1) :: ("a",2) :: ("a",3) :: []
= [("a",1); ("a",2); ("a",3)]

Q 3)

We have that
"a" : string , [1;2;3] : int list and
f's type can be instantiated to
f : ~~...~~ string -> int list -> (string * int) list
using 'a = string and 'b = int. Using the type rule for function application:
f "a" [1; 2; 3] : (string * int) list

Q 4)

f is <u>not</u> tail recursive because the recursive call in | y::ys -> (x,y) :: <u>f</u> x y
is <u>not</u> a tail call. When f x y returns a value res, the expression (x,y) :: res must still be evaluated.

Q 5)
```
let rec fA acc x = function
                     | []    -> List.rev acc
                     | y::ys -> fA ((x,y)::acc)
                                   x ys
```

Q 6)
```
let f x ys = List.map (fun y -> (x,y)) ys
```