1 Tail recursion and the fold functions

Last time there was some confusion about tail recursion and the fold function. Here are the fold_left and fold_right functions as they are defined in the List module.

```ocaml
let rec fold_left f accu l =  
  match l with  
  | []  -> accu  
  | a::l  -> fold_left f (f accu a) l

let rec fold_right f l accu =  
  match l with  
  | []  -> accu  
  | a::l  -> f a (fold_right f l accu);
```

On one level, there seems to be no difference between these functions.

```ocaml
# let intlist = [1;2;3;100;200];;
# fold_left (+) 0 intlist;;
- : int = 306

# fold_right (+) intlist 0;;
- : int = 306
```

In fact, they make the same calculation. However they calculate this value differently. Superficially, fold_left appears to add the numbers from left to right and fold_right from right to left. We can see this by asking our function to print statements explaining what it is doing.

```ocaml
let plus a b =  
  print_endline ((string_of_int a)^”+”^(string_of_int b));
  a+b;;

# fold_left plus 0 intlist;;
  0 + 1
  1 + 2
  3 + 3
  6 + 100
  106 + 200
- : int = 306

# fold_right plus intlist 0;;
  200 + 0
```
100 + 200
3 + 300
2 + 303
1 + 305
- : int = 306

Now let’s modify the fold functions to tell us when they are making recursive calls.

```ocaml
let rec fold_left f accu l =
  match l with
  | [] -> let () = print_endline ("at the bottom") in accu
  | a::l -> let () = print_endline ("digging down") in fold_left f (f accu a) l

let rec fold_right f l accu =
  match l with
  | [] -> let () = print_endline ("at the bottom") in accu
  | a::l -> let () = print_endline ("digging down") in f a (fold_right f l accu);;
```

Now let’s see what happens when we sum op the numbers in our intlist.

```ocaml
# fold_left plus 0 intlist ;;
digging down
0 + 1
digging down
1 + 2
digging down
3 + 3
digging down
6 + 100
digging down
106 + 200
at the bottom
- : int = 306

# fold_right plus intlist 0;;
digging down
```
digging down
digging down
digging down
digging down
at the bottom
200 + 0
100 + 200
3 + 300
2 + 303
1 + 305
- : int = 306

Notice that the `fold_left` function goes down the list and when it gets to the end, it has the final value. But the `fold_right` function goes down the list and then has to climb back out in order to calculate the value. This is a crucial difference. In `fold_right`, every time a recursive call is made, the computer has to remember to return to the earlier call. This takes more memory, and can slow things down. On the other hand, `fold_left` just digs down and finds the answer. It can throw away all the earlier calls as it does not need to return to them. Recursive functions which behave like this are called \textit{tail-recursive}.

So what about the fold function we wrote in lesson 3?

```ocaml
let rec fold f init_value list =
  match list with
  | [] -> init_value
  | h::t -> fold f (f h init_value) t;;
fold plus 0 int list;;
```

Let's modify it to tell us what it is doing.

```ocaml
let rec fold f init_value list =
  match list with
  | [] ->
    let () = print_endline "at the bottom" in
    init_value
  | h::t ->
    let () = print_endline "digging down" in
    fold f (f h init_value) t;;
fold plus 0 int list;;
```

So it is \textit{tail} recursive. The type signature does not match Ocaml's built-in \texttt{List.fold_left} because I passed in \texttt{(f h init_value)} instead of passing the arguments to \texttt{f} in the reverse order as is done in \texttt{List.fold_left: (f accu a)}. If we reverse this, we'll get the same type
signature as \texttt{List.fold\_left}.

```
let rec fold f init\_value list =
  match list with
  | [] -> init\_value
  | h::t -> fold f (f init\_value h) t;;
```

What makes \texttt{List.fold\_left} tail-recursive is that the application of \texttt{f} falls \textit{inside} the scope of the recursive call to \texttt{fold\_left}.

```
| a::l -> fold\_left f (f accu a) l
```

On the other hand, the recursive call falls within the scope of \texttt{f} in \texttt{fold\_right}.

```
| a::l -> f a (fold\_right f l accu);;
```

2 Important Lessons Today

1. Strategic placement of print command can help you understand what your code is doing.

2. When writing recursive functions, tail-recursion is preferred for its efficiency.

3. However, we are more interested in getting the program right than in efficiency. In fact, our primary concern is to make sure our program performs the function we want it to. Efficiency and optimization are secondary concerns. When programming: think about correctness first and efficiency later.