fold_left (+) 0 [3;4;5];;
fold_left (+) 0 [3;4;5];

match [3;4;5] with [] -> 0 | h::t -> fold_left (+) ((+) 0 h) t;
fold_left (+) 0 [3;4;5];
fold_left (+) ((+) 0 3) [4;5];
fold_left (+) 0 [3;45];
fold_left (+) 3 [45];
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 0 [3;4;5];;
fold_left (+) 3 [4;5];;
match [4;5] with [] -> 3 | h::t -> fold_left (+) ((+) 3 h) t;;
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) ((+) 3 4) [5];
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
fold_left (+) 0 [3;4;5];;
fold_left (+) 3 [4;5];;
fold_left (+) 7 [5];;

fold_left
  f = (+), b = 7, l = [5]

fold_left
  f = (+), b = 3, l = [4;5]
  h = 4, t = [5]

fold_left
  f = (+), b = 0, l = [3;4;5]
  h = 3, t = [4;5]
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
match [5] with [] -> 7 | h::t -> fold_left (+) ((+) 7 h) t;
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
fold_left (+) ((+) 7 5) [];

fold_left
f = (+), b = 7, l = [5]
h = 5, t = []

fold_left
f = (+), b = 3, l = [4;5]
h = 4, t = [5]

fold_left
f = (+), b = 0, l = [3;4;5]
h = 3, t = [4;5]
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
fold_left (+) 12 [];

fold_left
f = (+), b = 7, l = [5]
h = 5, t = []

fold_left
f = (+), b = 3, l = [4;5]
h = 4, t = [5]

fold_left
f = (+), b = 0, l = [3;4;5]
h = 3, t = [4;5]
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
fold_left (+) 12 [];

fold_left
f = (+), b = 12, l = []

fold_left
f = (+), b = 7, l = [5]
h = 5, t = []

fold_left
f = (+), b = 3, l = [4;5]
h = 4, t = [5]

fold_left
f = (+), b = 0, l = [3;4;5]
h = 3, t = [4;5]
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
fold_left (+) 12 [];
m(match [ ] with [ ] -> 12 | h::t -> fold_left (+) ((+) 12 h) t);
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
fold_left (+) 12 [];
12
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
12
fold_left (+) 0 [3;4;5]
fold_left (+) 3 [4;5]
12

<table>
<thead>
<tr>
<th>fold_left</th>
<th>f = (+), b = 3, l = [4;5]</th>
<th>h = 4, t = [5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>fold_left</td>
<td>f = (+), b = 0, l = [3;4;5]</td>
<td>h = 3, t = [4;5]</td>
</tr>
</tbody>
</table>
fold_left (+) θ [3;4;5];
12
fold_left (+) 0 [3;4;5];;
fold_left (+) 0 [3;4;5];;
match [3;4;5] with [] -> 0 | h::t -> fold_left (+) ((+) 0 h) t;
fold_left (+) 0 [3;4;5];;
fold_left (+) ((+) 0 3) [4;5];;
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 3 [4;5];
fold_left (+) 3 [4;5];

match [4;5] with [] -> 3 | h::t -> fold_left (+) ((+) 3 h) t;
fold_left (+) 3 [4;5];
fold_left (+) ((+) 3 4) [5];
fold_left (+) 3 [4;5];
fold_left (+) 7 [5];
fold_left (+) 3 [4;5];;
fold_left (+) 7 [5];;
fold_left (+) 7 [5];
fold_left (+) 7 [5];
match [5] with [] -> 7 | h::t -> fold_left (+) ((+) 7 h) t;
fold_left (+) 7 [5];
fold_left (+) ((+) 7 5) [];;
fold_left (+) 7 [5];
fold_left (+) 12 [];
fold_left (+) 7 [5];;
fold_left (+) 12 [];;
fold_left (+) 12 [];;
fold_left (+) 12 [];;
match [] with [] -> 12 | h::t -> fold_left (+) ((+) 12 h) t;;
fold_left (+) 12 [[]];
12
fold_left (+) 0 [3;4;5];;
fold_left (+) 0 [3;4;5];
match [3;4;5] with [] -> 0 | h::t -> fold_left (+) ((+) 0 h) t;
fold_left (+) 0 [3;4;5];
fold_left (+) ((+) 0 3) [4;5];
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
fold_left (+) 0 [3;4;5];
fold_left (+) 3 [4;5];
Tail Recursion

A recursive function is **tail recursive** iff the value returned by every recursive call is returned directly to the caller without being further acted upon or used in any subsequent computation.

```ocaml
let rec fold_left f b l =    match l with
  | [] -> b
  | h::t -> fold_left f (f b h) t;;
```
Tail Recursion

A recursive function is **tail recursive** iff the value returned by every recursive call is returned directly to the caller without being further acted upon or used in any subsequent computation.

```ml
let rec fold_left f b l =
  match l with
  | [] -> b
  | h::t -> fold_left f (f b h) t;;
```

*returned result of recursive call is returned to caller*
Example of a Non-tail recursive function

A recursive function is **tail recursive** iff the value returned by every recursive call is returned directly to the caller without being further acted upon or used in any subsequent computation.

```
let rec fold_right f l b =
    match l with
    | [] -> b
    | h::t -> f h (fold_right f (f t b));;
```
Example of a Non-tail recursive function

A recursive function is **tail recursive** iff the value returned by every recursive call is returned directly to the caller without being further acted upon or used in any subsequent computation.

```
let rec fold_right f l b =
  match l with
  | [] -> b
  | h::t -> f h (fold_right f (f t b));;
```

*this recursive call’s result is not directly returned, so this is not a tail-recursive function*
Tail Recursion

Note that tail recursive functions may have multiple recursive calls, and the recursive calls might not appear syntactically “last”. Think through the computation to decide whether a function is tail-recursive.

Exercise: Is this function tail-recursive?

```
let rec myfunc n =
  if n>10 then myfunc (n+2)
  else if n>0 then myfunc (n-1)
  else n;;
```