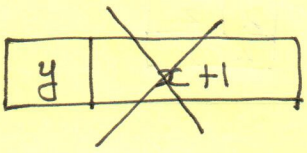
Twas brillig and the slithy toves did gyre and
gimbel in the wabe.

```
let rec rpe (b , e) =
    if b=0 then 0
    else if e=0 then 1
  else if (odd e) then
          b* rpe (b, e-1)
       else rpe(b, e/2) * rpe(b, e/2).
            let v = rpe (b, e/2)
            in  v * v
```

NOT TAIL RECURSIVE

let creates a binding
Bindings come in layers.

We never bind a name to an unevaluated expression.




```
let x = 1729 in
  let y = x+1 in
    x + y
```

y	1730
x	1729

```
let x = 1 in
  let y = x+1 in
    let x = 2 in
      y
```

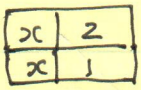
⇝ 2

NOT 3

x	2
y	2
x	1

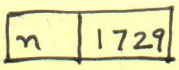
The meaning of a binding does not change: STATIC BINDING

```
let x = 1 in
 let  x = 2  in
      ~<exp>.
```



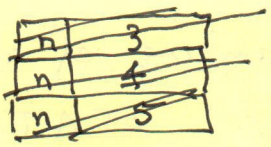

```
let inc (n) =  n+1
```
→ parameter

```
inc (1729)
```
→ argument



← a binding is created by a function application (or call) when the call is over this binding is removed.


```
inc 3;;
inc 5;;
inc (inc (inc 3));;
```



→ 6

:: constructor

hd } destructors extractors
tl }

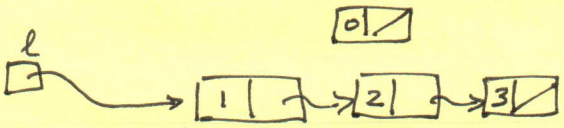
instead of destructors use patterns

$0 :: [1; 2; 3] \rightsquigarrow [0; 1; 2; 3]$

What if I want $[1; 2; 3; 0]$?

Cannot write $[1; 2; 3] :: 0$ ✗ type error

$:: \quad \alpha \times \alpha\text{-list} \to \alpha\text{-list}$



We have to write code to put an item at the back of a list.

let rec append $(l_1, l_2) \rightsquigarrow$ $l_1 @ l_2$ concatenated

append $([1; 2; 3], [4; 5; 6]) \rightsquigarrow [1; 2; 3; 4; 5; 6]$

$[\,]$ → empty list

```
let rec append (l1, l2) =
    match l1 with
    | [] -> l2
    | x::xs -> x::(append (xs, l2))
```

O(n)

```
    match <this> with
    | <this-pattern> -> <return this>
    | ...
```

$[1; 2; 3]$

$1 :: [2; 3]$

$x :: xs$

:: constructor

hd, tl } destructors extractors

instead of destructors use patterns

$0 :: [1; 2; 3] \rightsquigarrow [0; 1; 2; 3]$

What if I want $[1; 2; 3; 0]$?

Cannot write $[1; 2; 3] :: 0$ ✗ type error

:: $\alpha \times \alpha\text{-list} \to \alpha\text{-list}$

We have to write code to put an item at the back of a list.

let rec append $(l_1, l_2) \rightsquigarrow$ $l_1 @ l_2$ (concatenated)

append $([1; 2; 3], [4; 5; 6]) \rightsquigarrow [1; 2; 3; 4; 5; 6]$

[ ] → empty list

```
let rec append (l1, l2) =
    match l1 with
    | [] -> l2
    | x::xs -> x::(append (xs, l2))
```

O(n)

$[1; 2; 3]$

$1 :: [2; 3]$

$x :: xs$

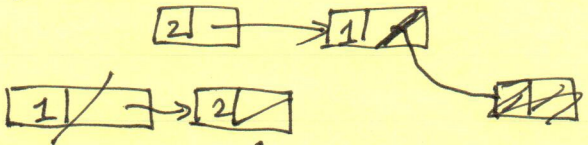
```
    match <this> with
    | <this-pattern> -> <return this>
    | - - -
```

```
let rec reverse l =
    match l with
  | [] -> []                                  O(n^2)
  | x::xs -> (reverse(xs) @ [x])
```

```
let rev l =
    let rec helper (l, acc) =
        match l with
      | [] -> acc
      | x::xs -> helper(xs, x::acc)
    in
  helper (l, [])
```

one recursive call OUTERMOST

O(n)



zip ([1; 3; 5], [2; 4; 6]) ⇝ [1; 2; 3; 4; 5; 6]

```
let rec zip (l1, l2) =
    match l1 with
  | [] -> l2
  | x::xs ->   x::(zip(l2, xs))
```