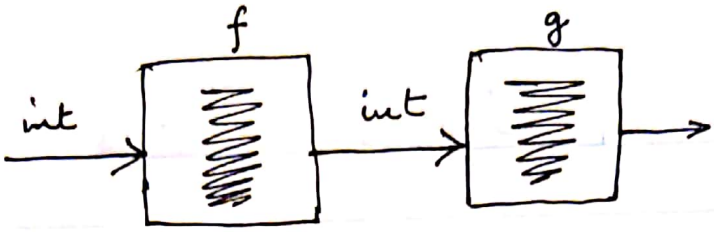


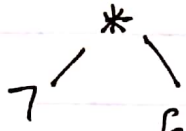
for a given input the output is unique

$g \circ f$ → follows standard mathematical rules for function composition

---

```
let rec fact n =
    if n=0 then 1
    else n * fact (n-1)
```

fact (7)



many pending computations.

```
let fastfact n =
    let rec helper (n, m) =
        if n=0 then m
        else helper (n-1, n*m)
in
helper (n, 1)
```

Tail recursion

~~helper (7, 1)~~
~~helper (6, 7)~~
helper (5, 42)

name	value

binding

Environment consists of layers of such bindings

```
let x = 17 in
    let x = 5 in
        let x = 2 in
```

x

x	2
x	5
x	17

```
let rec fib n =                    0 1 1 2 3 5 8 13
    if n=0 then 0                  21 34 55 ...
    else if n=1 then 1
    else fib(n-1) + fib(n-2)

let tailfib n =
    let rec helper (n, a, b) =
        if n=0 then a
        else if n=1 then b
        else helper (n-1, b, a+b)
    in
helper (n, 0, 1)
```

- ONLY 1 recursive call
- Recursive call must be outermost

~~let rec fib = fun n ->~~

```
let rec sumnums lo hi =
    if lo > hi then 0
    else lo + sumnums (lo+1) hi

let tailsum lo hi =
    let rec helper lo hi tally =
        if lo > hi then tally
        else helper (lo+1) hi (tally+lo)
    in
    helper lo hi 0
```

```
let myadd (n, m) = n + m ;;
let myadd2 n m = n + m ;;
```

(,) : package two values as a single entity called a pair.

type of myadd int * int → int

(,) allows even two values of different type

(7, true) : int * bool

myadd 5 ;; → type error

myadd2 : int → (int → int)

let foo = myadd 2 5 → a function

foo : int → int

foo 3

8 : int

myadd2 5 3 → 8

CURRYING → HASKELL CURRY

Pattern matching: A parameter can be a "structured" name or a pattern and OCaml provides pattern matching as a way of taking structures apart.

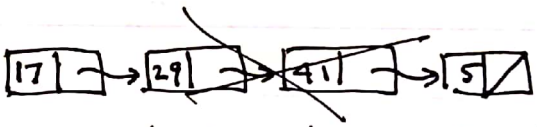
```
let myadd (n,m) =
    match (n,m) with
    | (0,_) -> x
    | (_,0) -> n
    |   _   -> n+m
```

Lists : Built-in structure

:: 'a * 'a list → 'a list

hd : 'a list → 'a

tl : 'a list → 'a list



int list.

Construction & destruction of lists is based on pointer manipulations. It does not matter what is stored in the cells.

'a list is a polymorphic type
↳ type parameter - type variable

17 :: [29;41;5]
x :: xs

[17; 29; 41; 5]

[17; 29; 41; 5] int list

3 :: [1; 5; 7] ⇝ [3; 1; 5; 7]

[ ] → empty list

```
let rec sumlist l =
  match l with
  | [ ] -> 0
  | x::xs -> x + (sumlist xs)
```