Programming Languages and Environments (Lecture 6)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Syllabus

- Recursive functions over integers
- Inductive Reasoning vs Iterative Reasoning
- Tail recursion

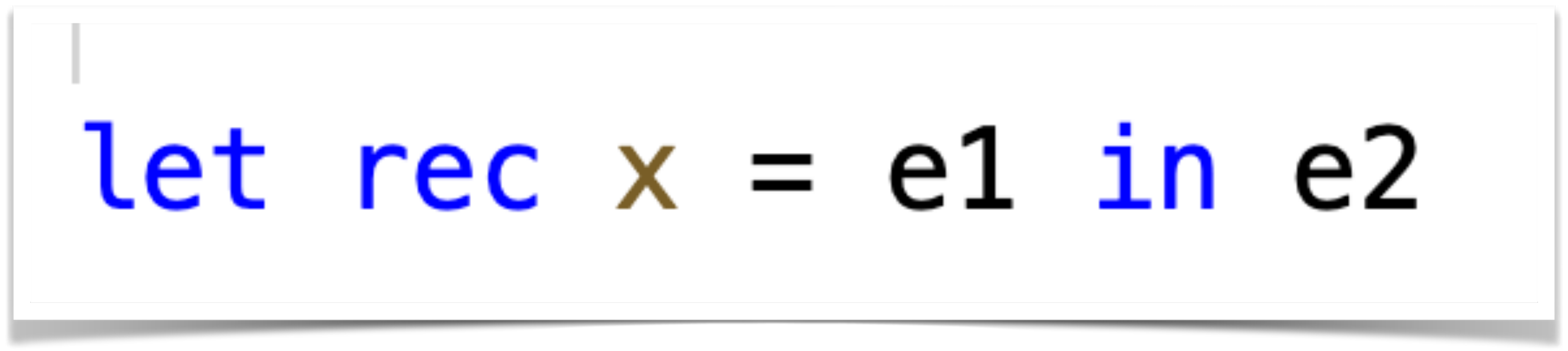
Type inference (Recap Exercise)

- How many solutions are there for the types of these expressions?

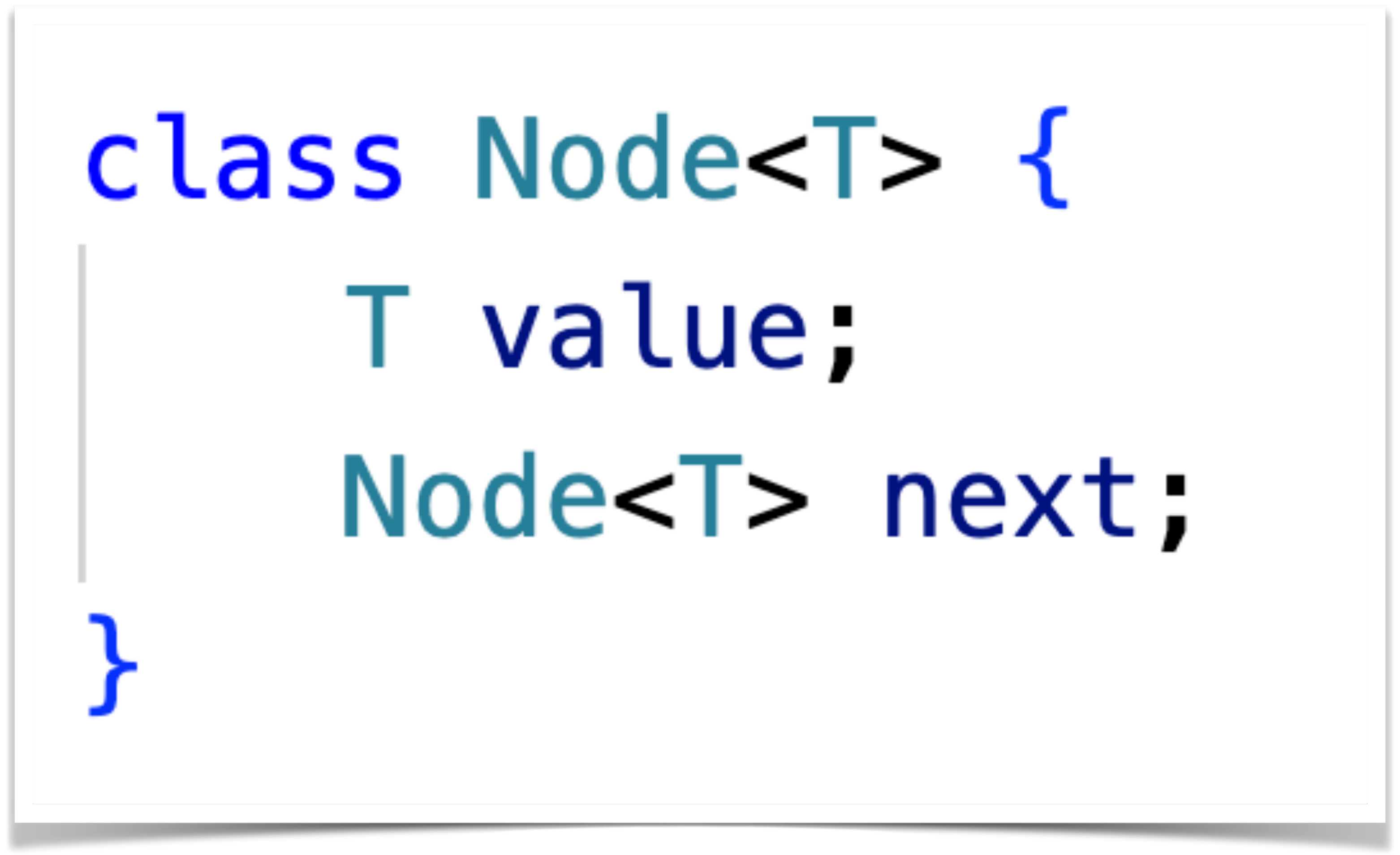
```
fun x y z -> if x then y else z

fun w -> if w then
             fun x y -> x = y
           else
             fun x y -> x <> y
```

Recursive functions

- Recursion is a mechanism that allows defining an entity (function, type, class, etc.) using its own name in its definition.



```
let rec x = e1 in e2
```



```
class Node<T> {
     T value;
     Node<T> next;
}
```

- Recursion is a mechanism that allows instantiating the same code (function) more than once in the same execution trace, with potentially different values for the parameters.

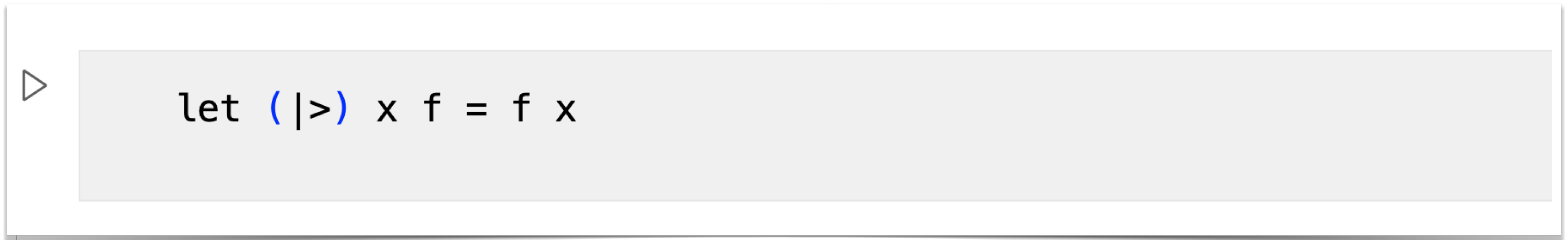
Recursive functions

- Functional languages use recursion as the main iteration mechanism.

```
let rec loop () = read_int () |> print_int; print_endline ""; loop ();;
```

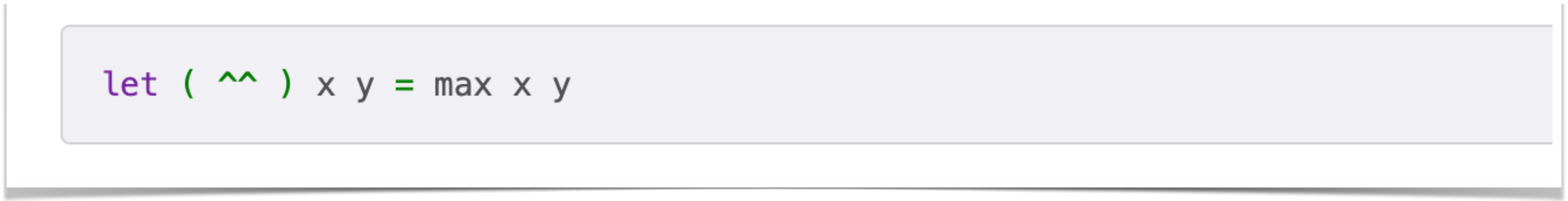
- The recursive use of a name must always be "guarded" by the definition of a function.
- Generally speaking, recursion requires maintaining a stack, which consumes space in a linear amount to the depth of recursion.

Syntax Hint: Operators

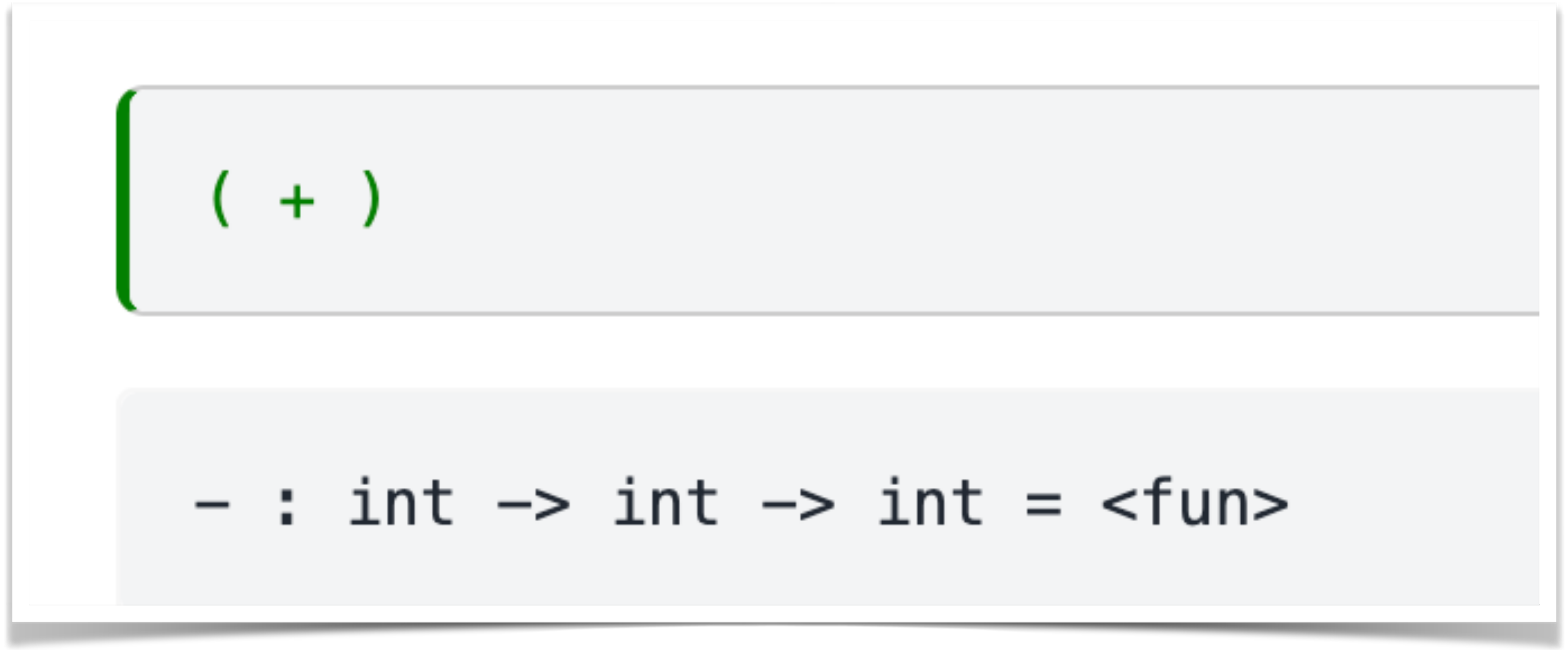


```
let (|>) x f = f x
```

```
let rec loop () = read_int () |> print_int; print_endline ""; loop ();;
```



```
let ( ^^ ) x y = max x y
```



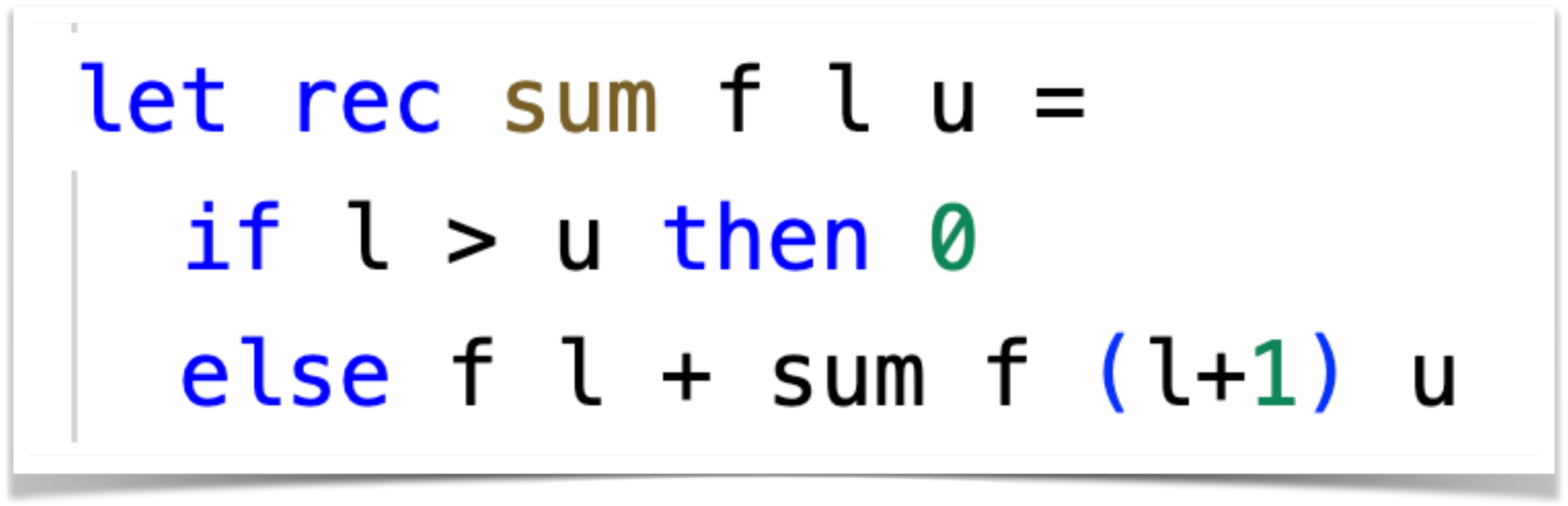
```
( + )
```

```
- : int -> int -> int = <fun>
```

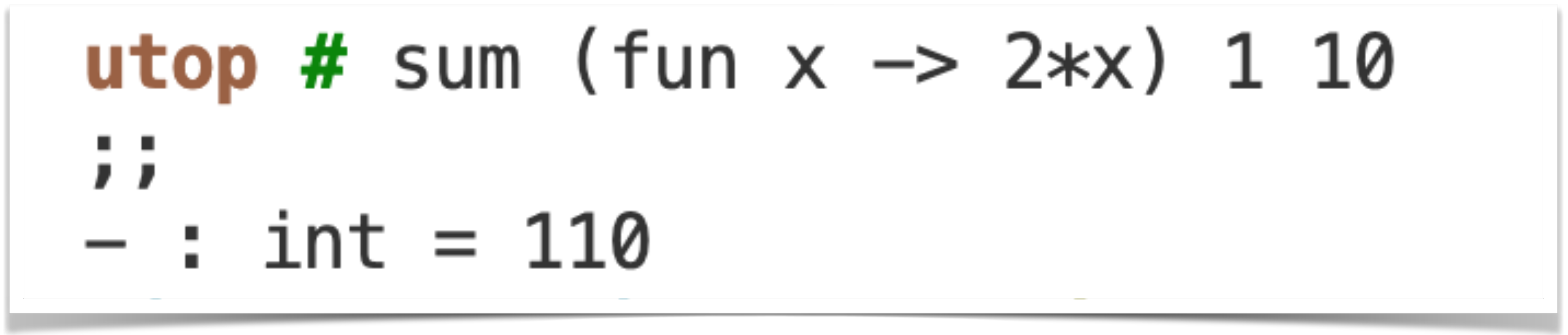
Recursive functions (General Sum)

- The general form of the sum is defined by a function that iterates from a lower bound (`l`) to an upper bound (`u`) applying a function (`f`). This approach, employing higher-order, is able to abstract the way elements are produced.

$$\sum_{i=l}^{u} f(i)$$



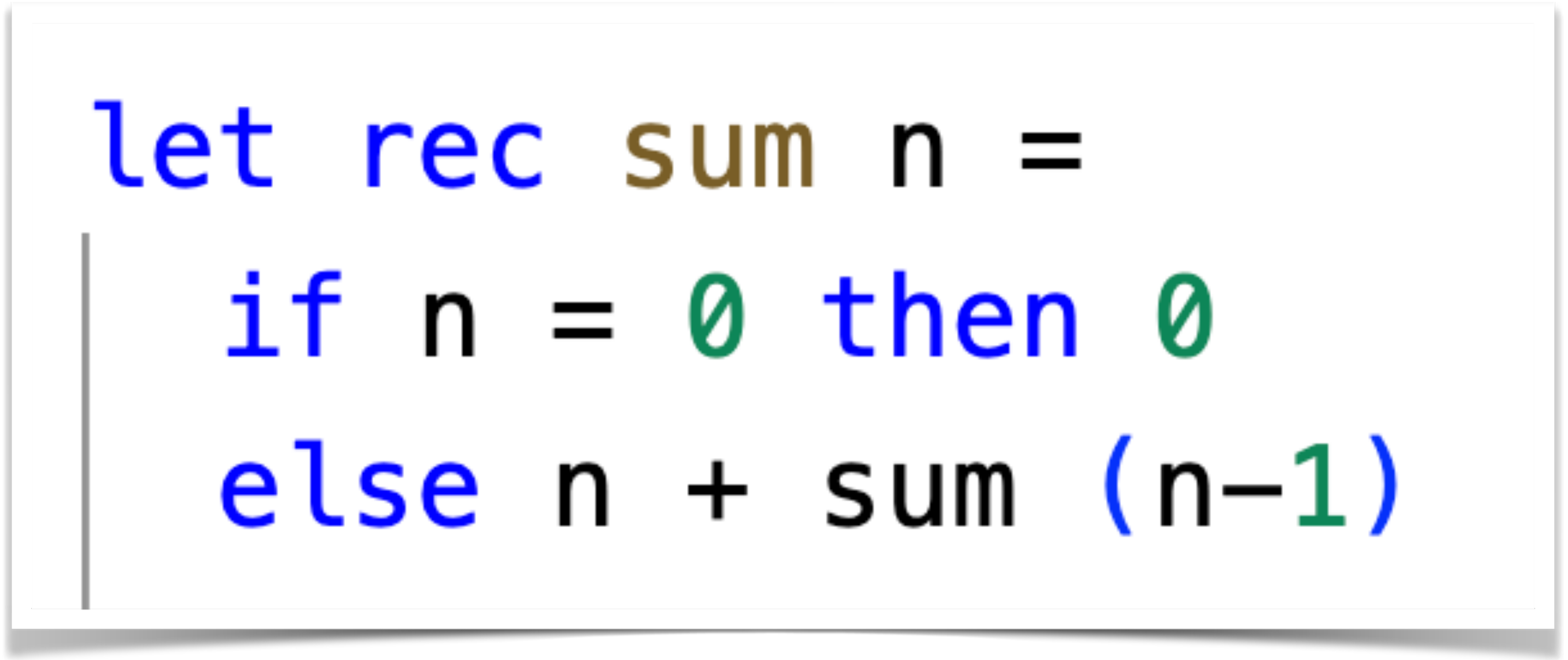
```
let rec sum f l u =
  if l > u then 0
  else f l + sum f (l+1) u
```



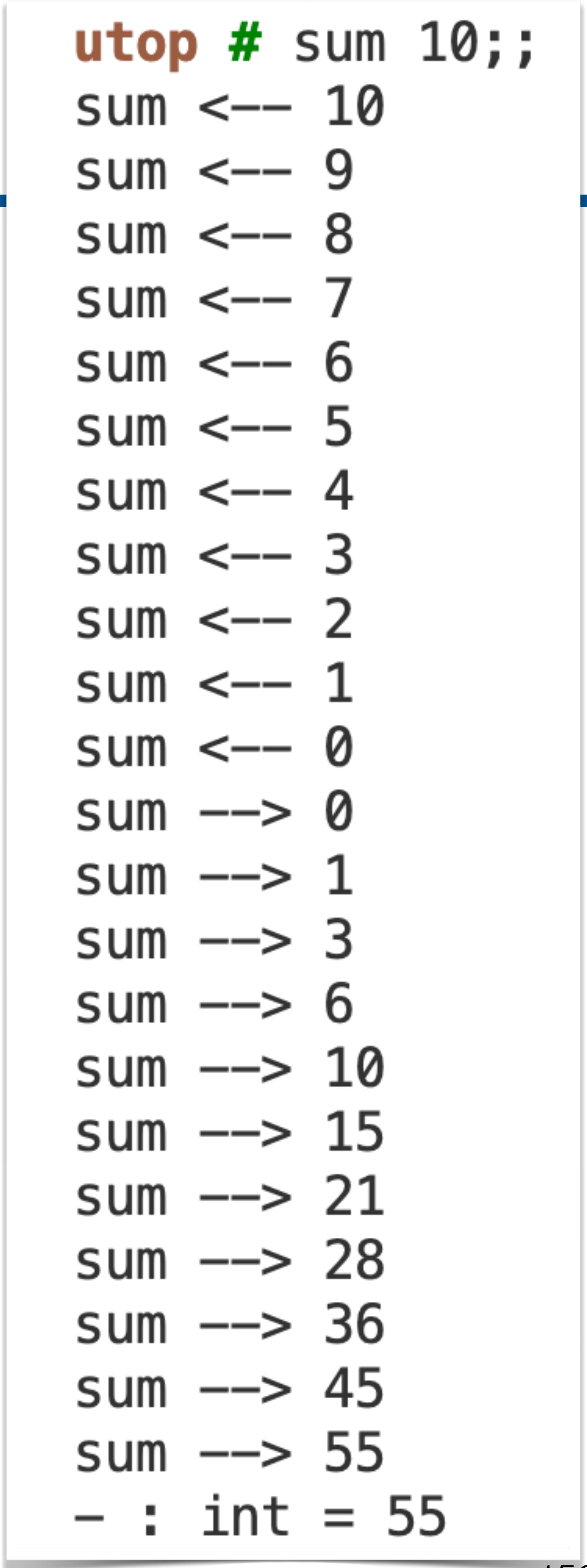
```
utop # sum (fun x -> 2*x) 1 10
;;
- : int = 110
```

Stack based execution

- Function calls are based on a stack, where parameter values for each call and the corresponding local variables are stored.
- The stack grows linearly with the depth of recursion.



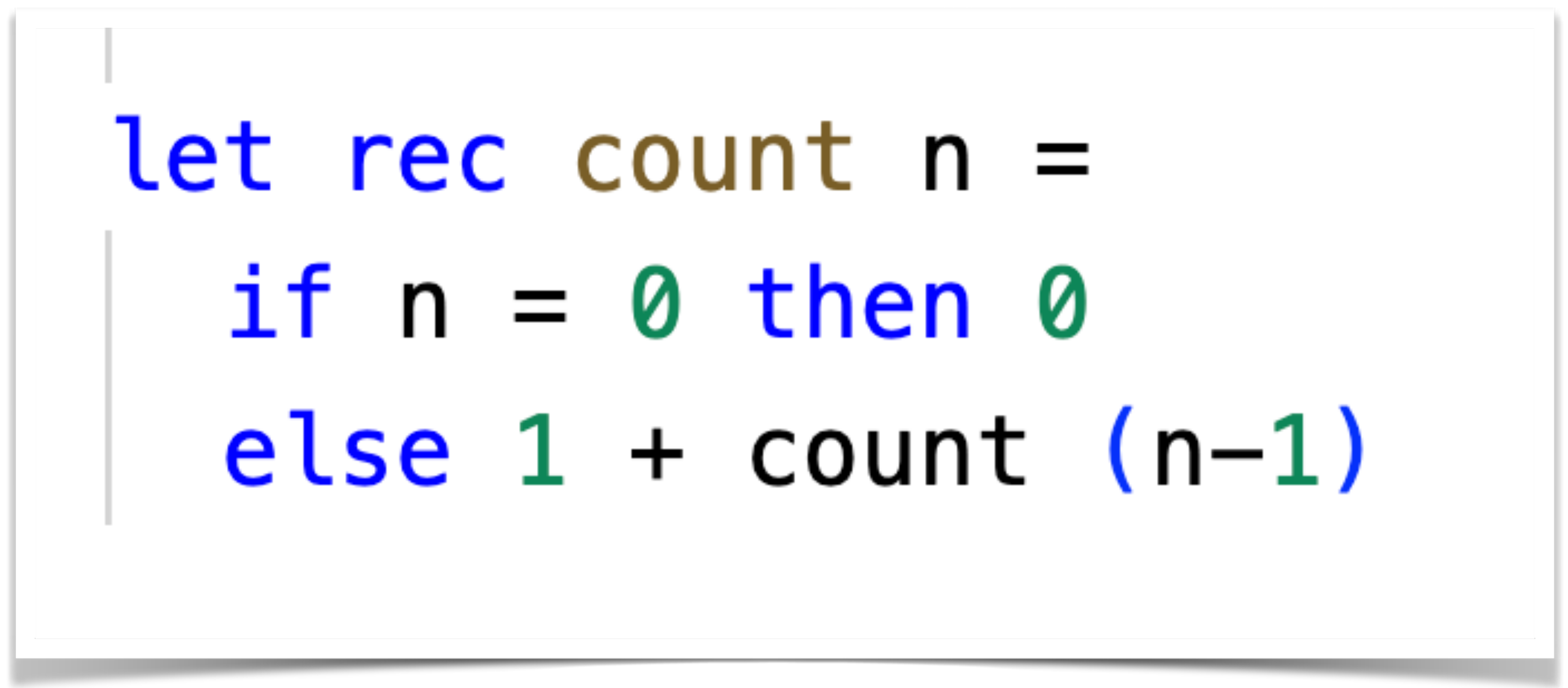
```
let rec sum n =
  if n = 0 then 0
  else n + sum (n-1)
```



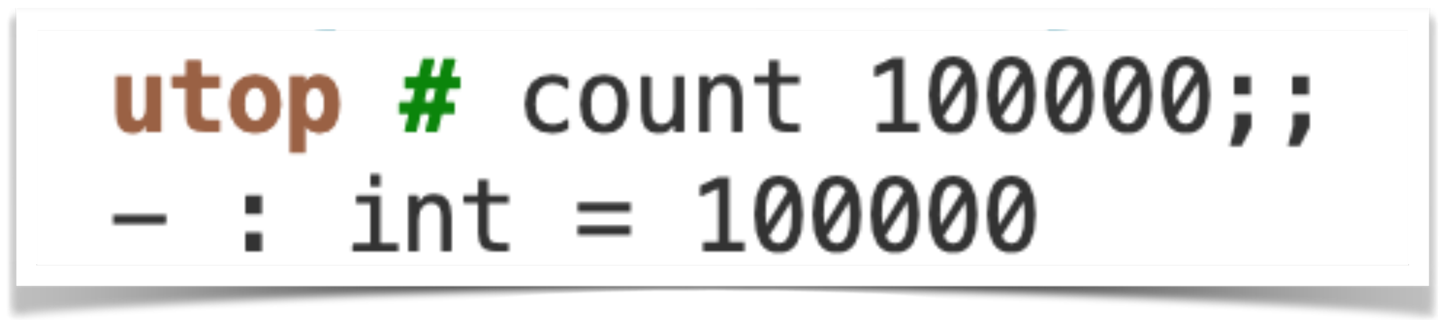
```
utop # sum 10;;
sum <-- 10
sum <-- 9
sum <-- 8
sum <-- 7
sum <-- 6
sum <-- 5
sum <-- 4
sum <-- 3
sum <-- 2
sum <-- 1
sum <-- 0
sum --> 0
sum --> 1
sum --> 3
sum --> 6
sum --> 10
sum --> 15
sum --> 21
sum --> 28
sum --> 36
sum --> 45
sum --> 55
- : int = 55
```

Stack based execution

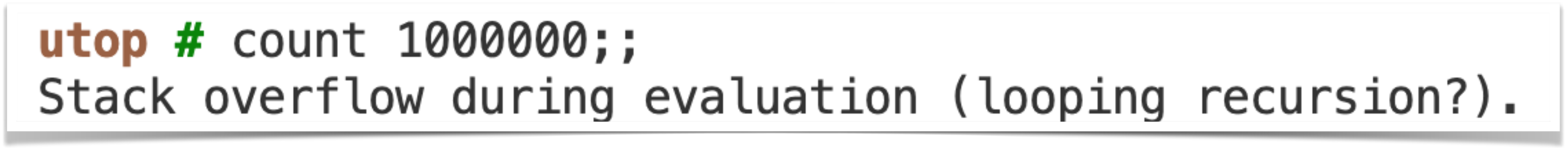
- Function calls are based on a stack, where parameter values for each call and the corresponding local variables are stored.
- The stack grows linearly with the depth of recursion.



```ocaml
let rec count n =
  if n = 0 then 0
  else 1 + count (n-1)
```



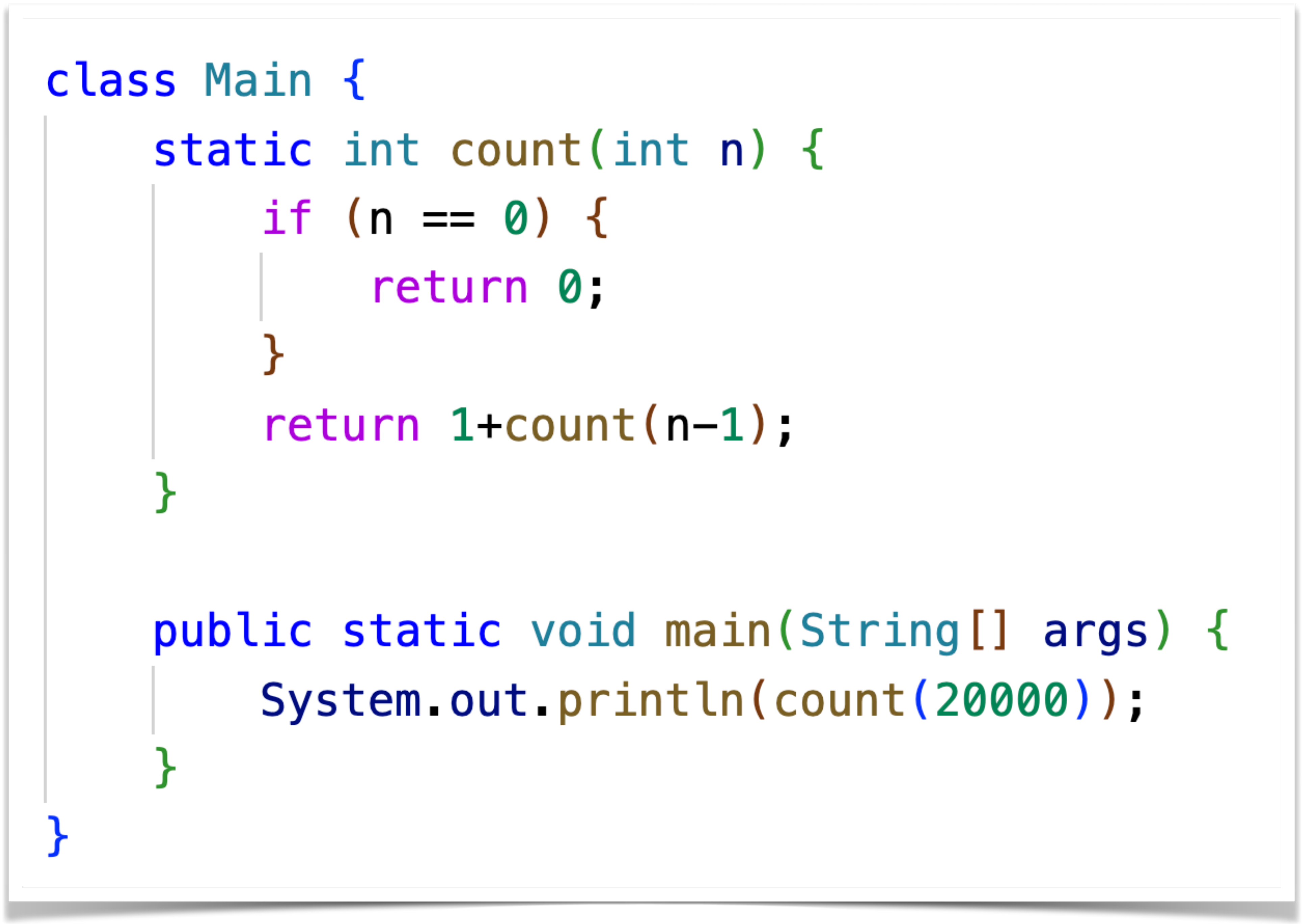
```
utop # count 100000;;
- : int = 100000
```



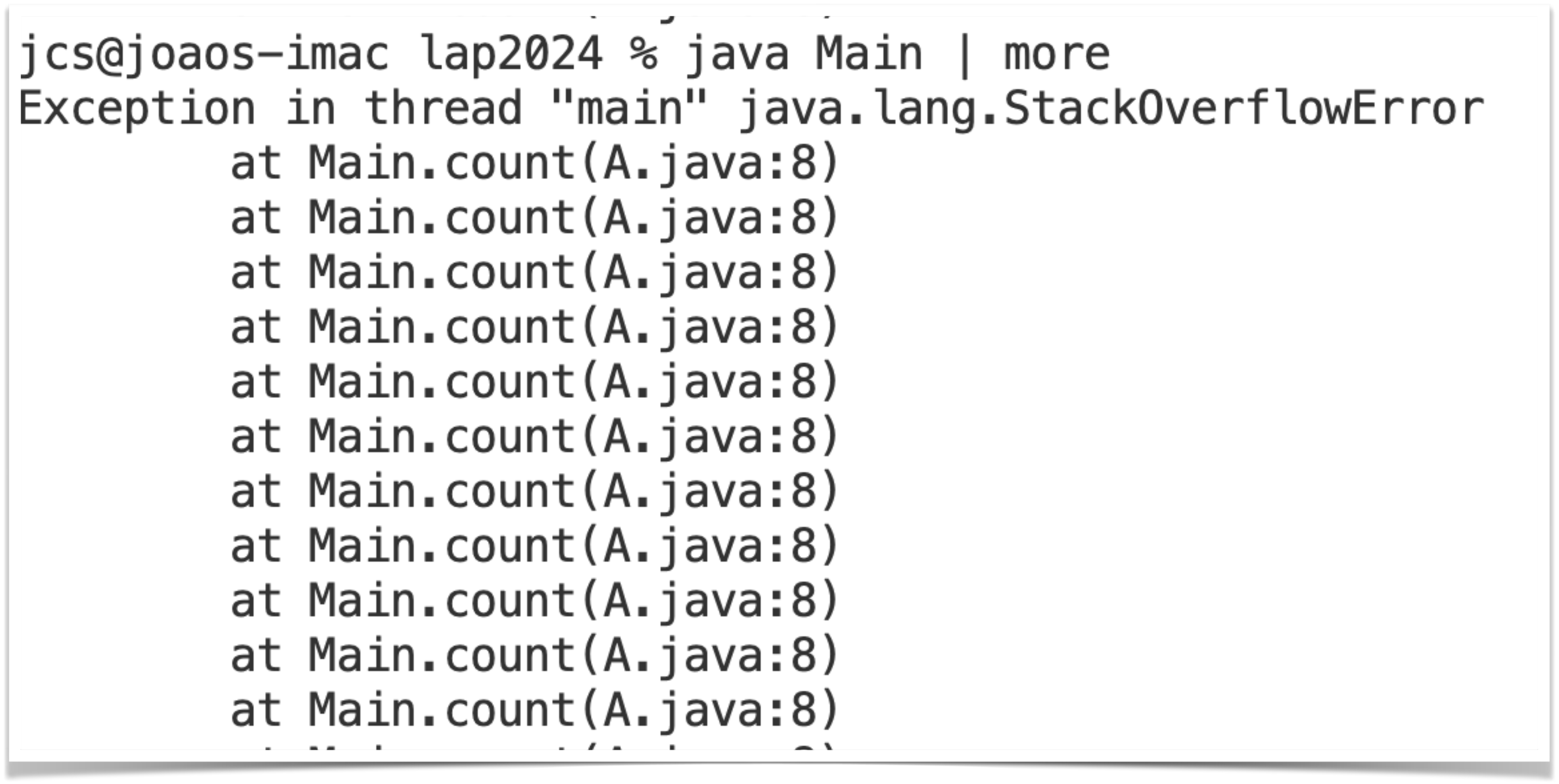
```
utop # count 1000000;;
Stack overflow during evaluation (looping recursion?).
```

Stack based execution

- Function calls are based on a stack, where parameter values for each call and the corresponding local variables are stored.
- The stack grows linearly with the depth of recursion.



```java
class Main {
    static int count(int n) {
        if (n == 0) {
            return 0;
        }
        return 1+count(n-1);
    }

    public static void main(String[] args) {
        System.out.println(count(20000));
    }
}
```



```
jcs@joaos-imac lap2024 % java Main | more
Exception in thread "main" java.lang.StackOverflowError
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
        at Main.count(A.java:8)
```

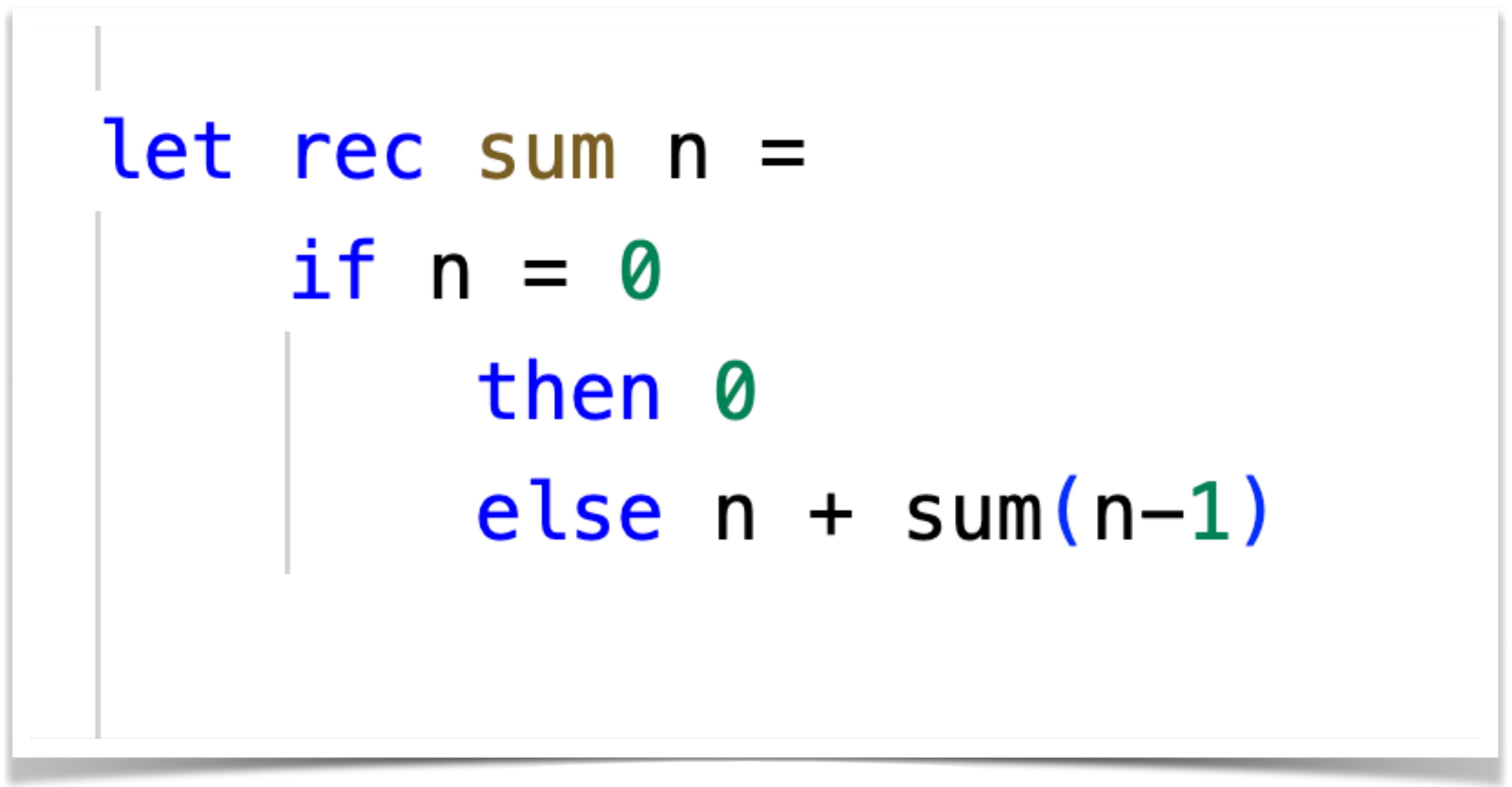
Execution stack and function calls

- When a function has some computation to perform between the recursive call and the end, and needs to return the result, then it needs to maintain all activation records.

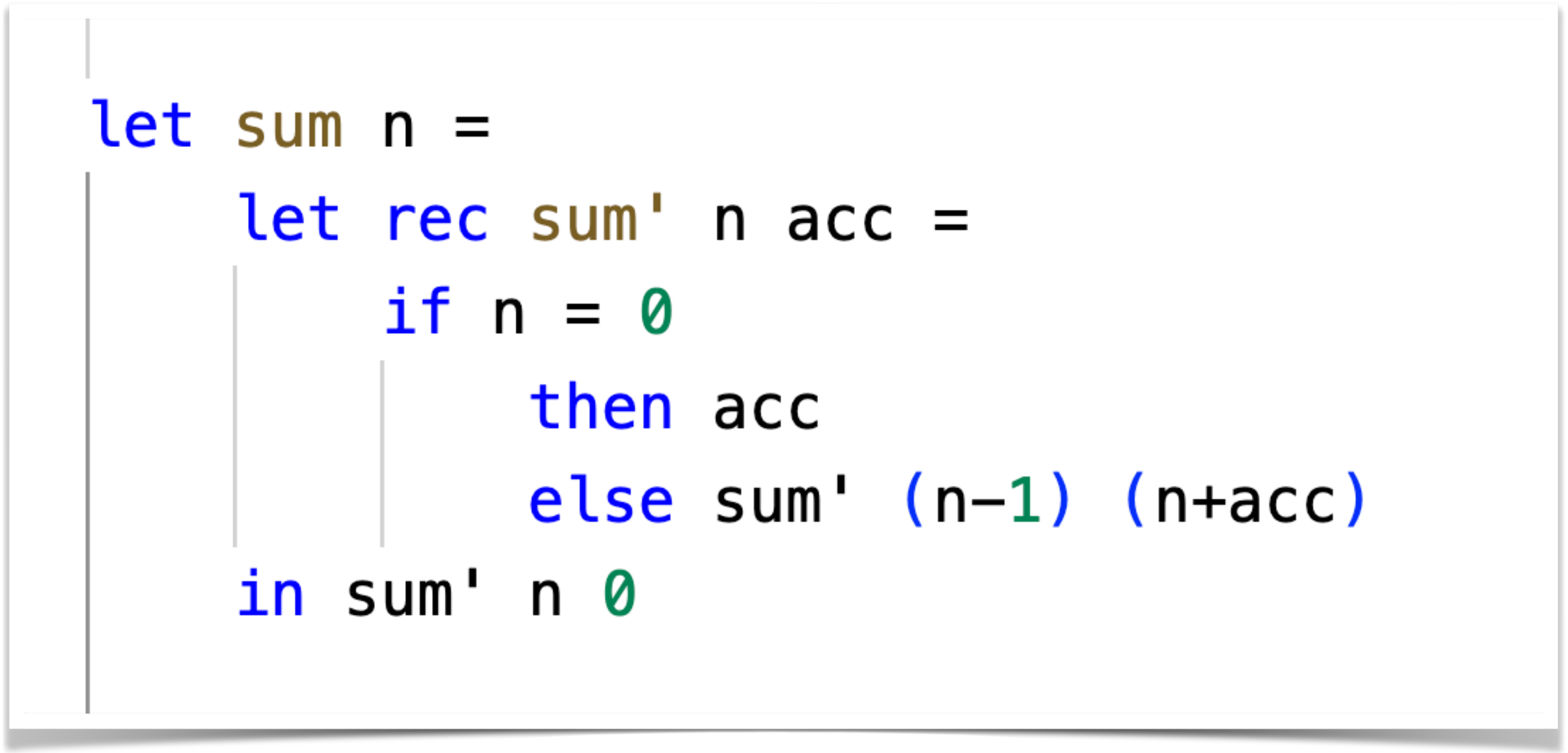
>sum 4
>sum 3
>sum 2
>sum 1
>sum 0>
sum 1>
sum 2>
>sum 3
>sum 4
n = 4
n = 3
n = 2
n = 1
n = 0
res = 0
res = 1
res = 3
res = 6
res = 10

“Tail recursion”

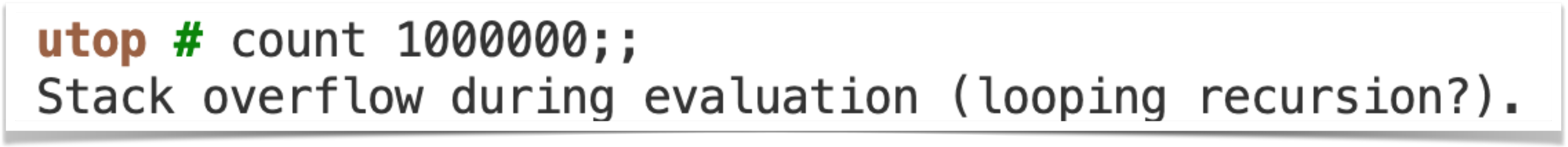
- The same function can be written in another way so that, after the recursive call, there is no more computation to be done.



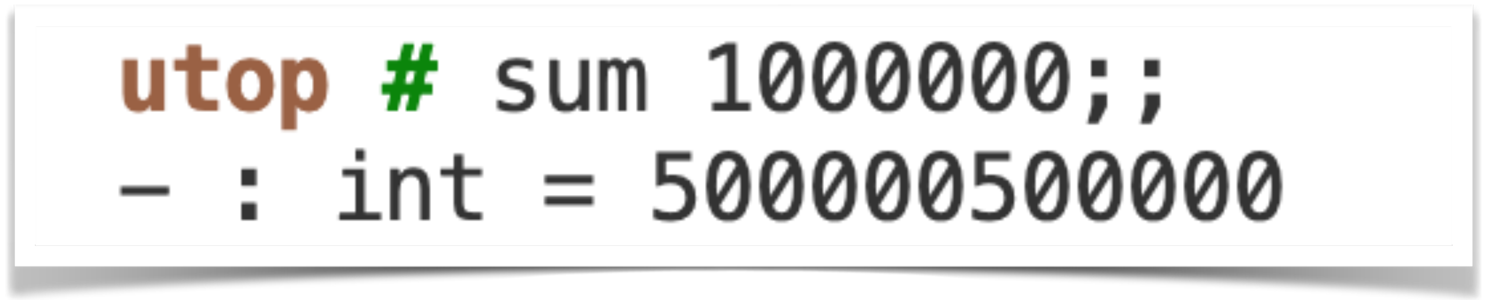
```
let rec sum n =
    if n = 0
        then 0
        else n + sum(n-1)
```



```
let sum n =
    let rec sum' n acc =
        if n = 0
            then acc
            else sum' (n-1) (n+acc)
    in sum' n 0
```



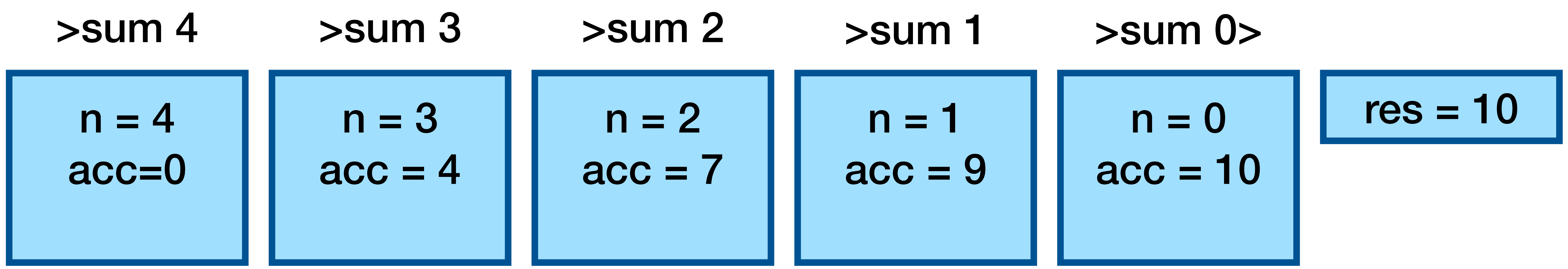
```
utop # count 1000000;;
Stack overflow during evaluation (looping recursion?).
```



```
utop # sum 1000000;;
- : int = 500000500000
```

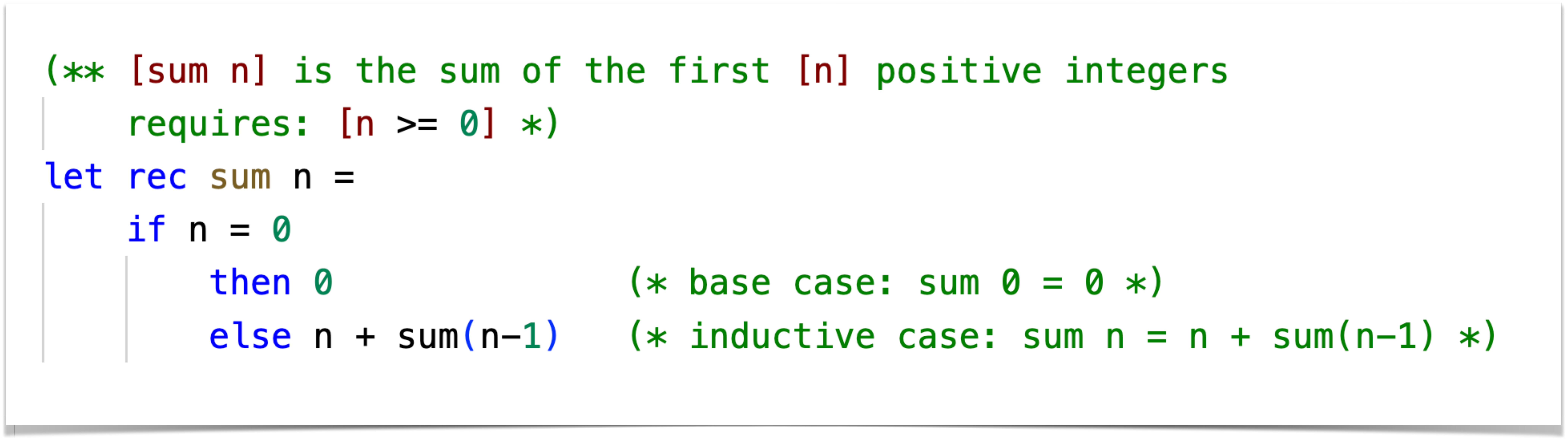
“Tail recursion” and the execution stack

- If the recursive call is the last thing to be done in the function, the activation record can be reused because the local variables will no longer be needed after returning, and the result is already in place (at the top of the stack).



Inductive functions over natural numbers

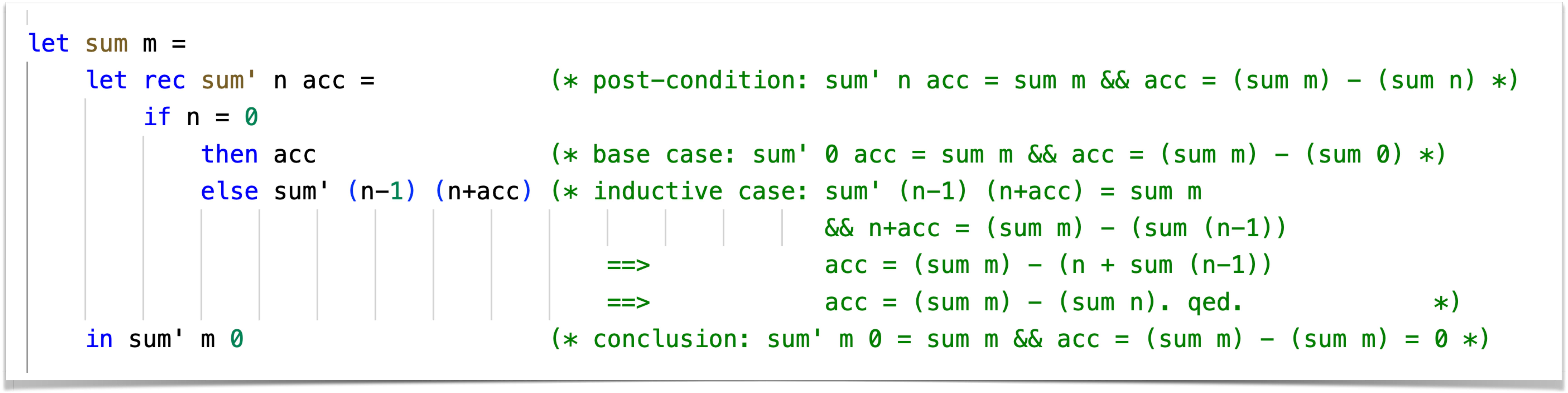
- Recursive functions can follow an inductive reasoning process to reach the result. It is possible to prove their correctness using an induction hypothesis.



```
(** [sum n] is the sum of the first [n] positive integers
    requires: [n >= 0] *)
let rec sum n =
    if n = 0
        then 0                     (* base case: sum 0 = 0 *)
        else n + sum(n-1)          (* inductive case: sum n = n + sum(n-1) *)
```

Inductive functions over natural numbers (tail recursion)

- Recursive functions can follow an inductive reasoning process to reach the result. It is possible to prove their correctness using an induction hypothesis.



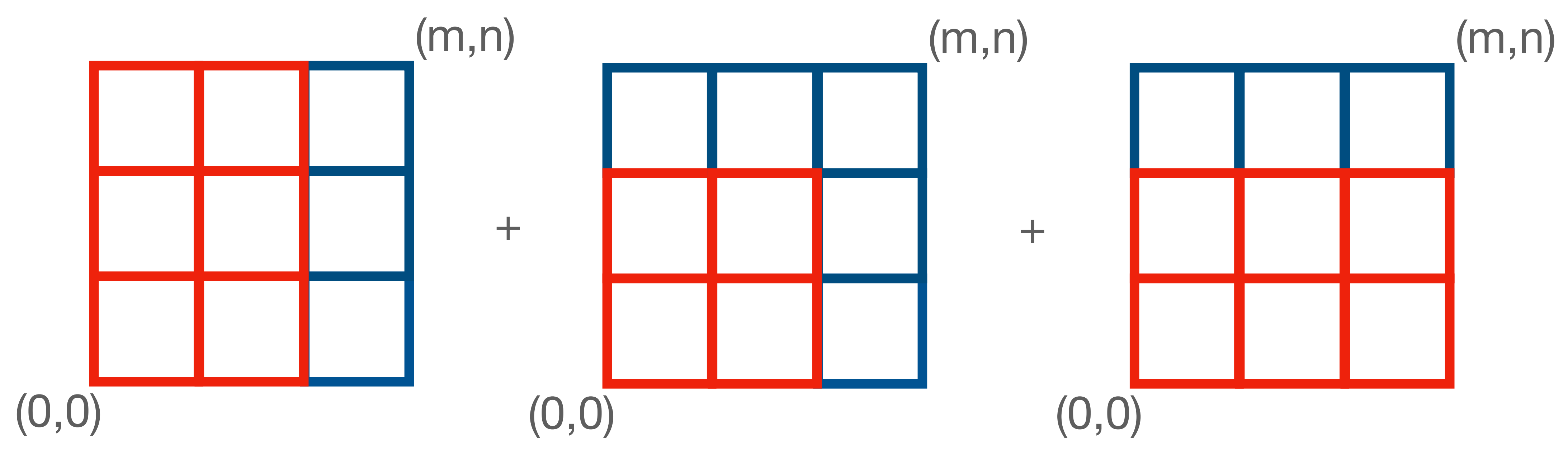
```
let sum m =
    let rec sum' n acc =                (* post-condition: sum' n acc = sum m && acc = (sum m) - (sum n) *)
        if n = 0
            then acc                    (* base case: sum' 0 acc = sum m && acc = (sum m) - (sum 0) *)
            else sum' (n-1) (n+acc) (* inductive case: sum' (n-1) (n+acc) = sum m
                                                       && n+acc = (sum m) - (sum (n-1))
                                        ==>            acc = (sum m) - (n + sum (n-1))
                                        ==>            acc = (sum m) - (sum n). qed.            *)
    in sum' m 0                         (* conclusion: sum' m 0 = sum m && acc = (sum m) - (sum m) = 0 *)
```

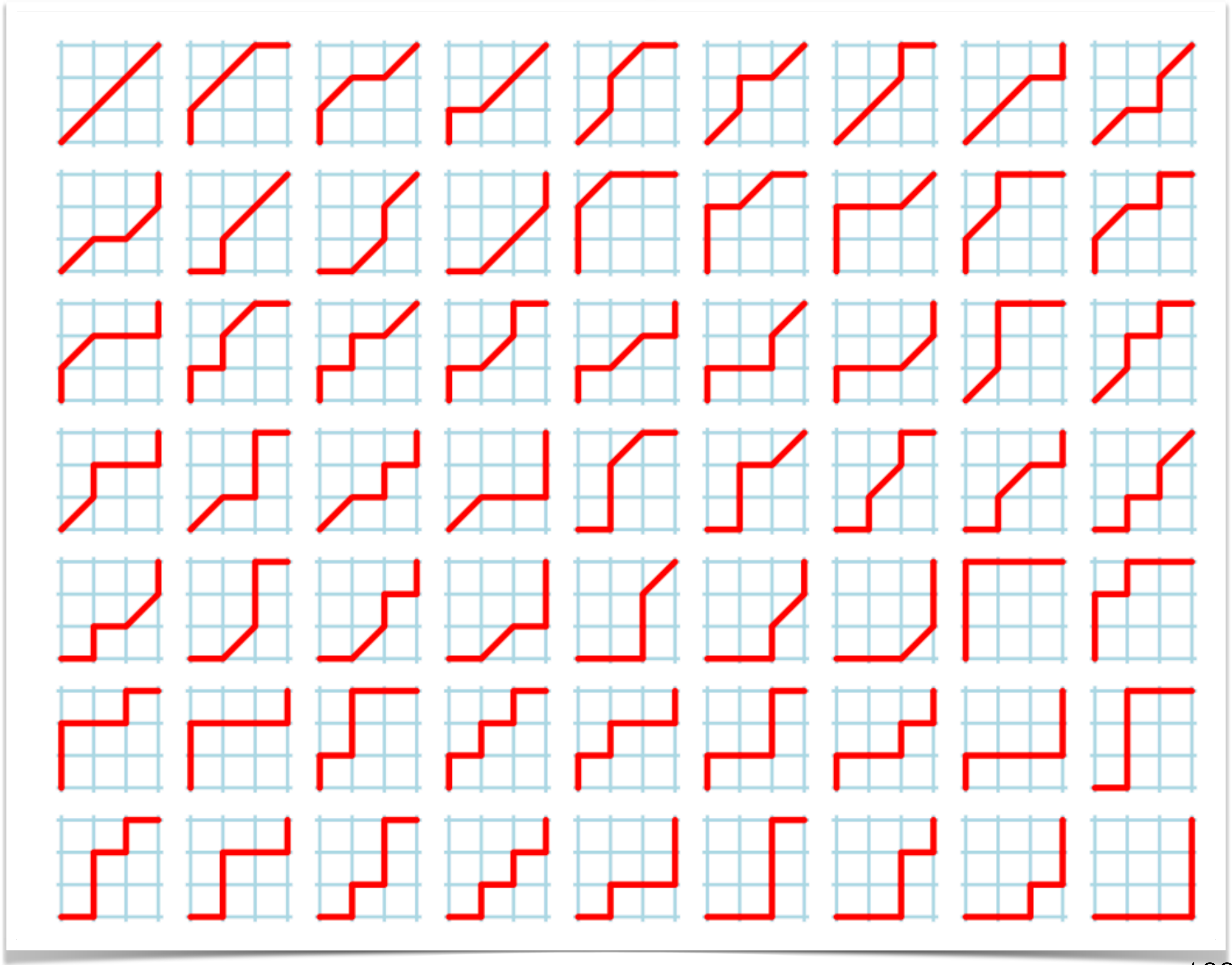
Inductive functions: Delannoy number

- Determine the number of paths that exist in a grid `n` by `m`, between the point `(0,0)` and the point `(m,n)`, using steps of length 1 in the **north**, **northeast** and **east** directions.

(0,0)

- 1

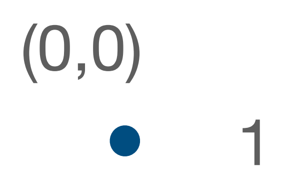


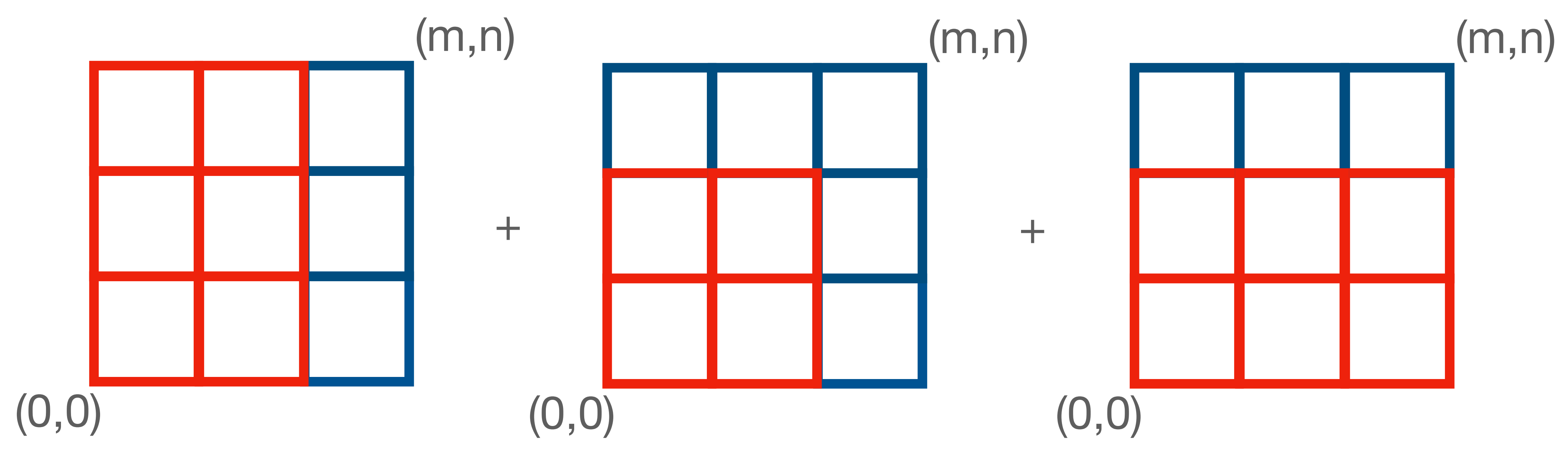


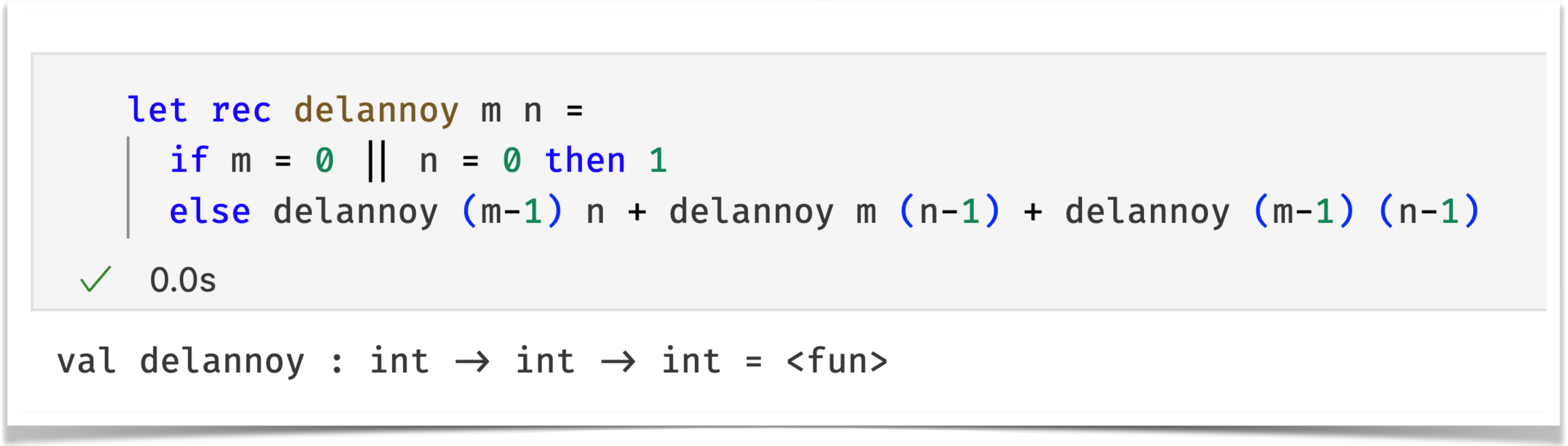
https://en.wikipedia.org/wiki/Delannoy_number

Inductive functions: Delannoy number

- Determine the number of paths that exist in a grid **n** by **m**, between the point `(0,0)` and the point `(m,n)`, using steps of length 1 in the **north**, **northeast** and **east** directions.



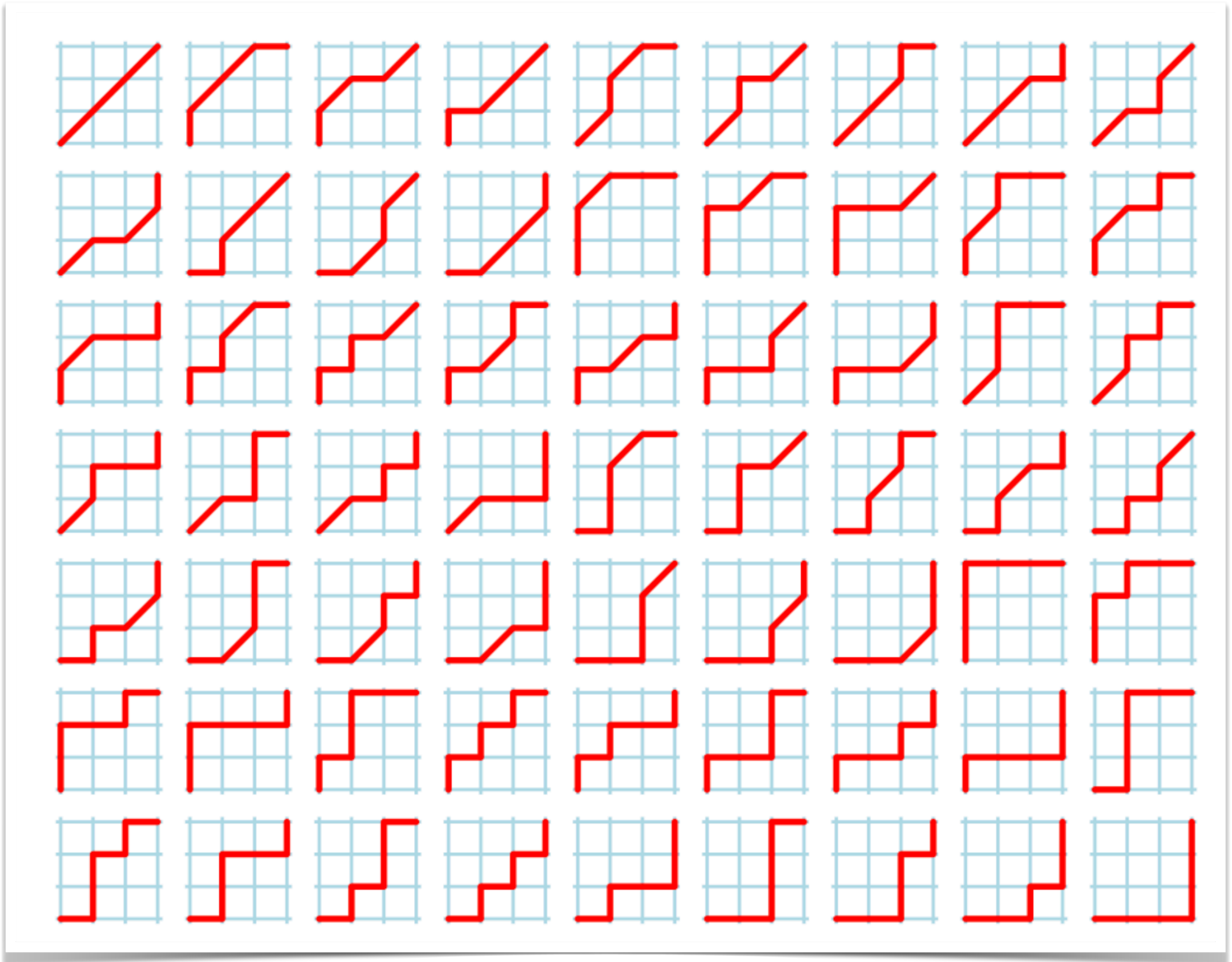




```
let rec delannoy m n =
  if m = 0 || n = 0 then 1
  else delannoy (m-1) n + delannoy m (n-1) + delannoy (m-1) (n-1)
```

✓ 0.0s

```
val delannoy : int → int → int = <fun>
```



Programming Languages and Environments (Lecture 7)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Syllabus

- User defined types: pairs, sum types, and pattern matching.