IT UNIVERSITY OF COPENHAGEN

Functional Programming

Patrick Bahr

Iterative functions and optimisations

Based on original slides by Michael R. Hansen

4 Weeks in: What you learned so far

- Core principles of functional programming:
  - (Recursive) Algebraic data types
  - Higher-order and recursive functions
  - Pattern matching
  - Polymorphic types and functions

First 3 weeks

IT UNIVERSITY OF COPENHAGEN

4 Weeks in: What you learned so far

- Core principles of functional programming:
  - (Recursive) Algebraic data types
  - Higher-order and recursive functions
  - Pattern matching
  - Polymorphic types and functions

First 3 weeks

- F# specific features:
  - Module system and interface files
  - Imperative features (loops, references)

Last week

This week

1. The memory model of F#
2. Example: Recursive list functions
3. Tail-recursive functions (aka. iterative functions)
   A. Using Accumulators
   B. Using Continuations

IT UNIVERSITY OF COPENHAGEN

Part I

The Memory Model of F#

The factorial function

Consider the factorial function:

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

- Computation time?
- Memory requirements?

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

The factorial function

fact 5 ⇝

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

fact 5 ⇝

5 * fact 4 ⇝

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

fact 5 ⇝

5 * fact 4 ⇝

5 * (4 * fact 3) ⇝

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

```
fact 5 ⇝
5 * fact 4 ⇝
5 * (4 * fact 3) ⇝
5 * (4 * (3 * fact 2)) ⇝
```

IT UNIVERSITY OF COPENHAGEN

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

```
fact 5 ⇝
5 * fact 4 ⇝
5 * (4 * fact 3) ⇝
5 * (4 * (3 * fact 2)) ⇝
5 * (4 * (3 * (2 * fact 1))) ⇝
```

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

```
fact 5 ⇝
5 * fact 4 ⇝
5 * (4 * fact 3) ⇝
5 * (4 * (3 * fact 2)) ⇝
5 * (4 * (3 * (2 * fact 1))) ⇝
5 * (4 * (3 * (2 * (1 * fact 0)))) ⇝
```

IT UNIVERSITY OF COPENHAGEN

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

fact 5 ⤳
5 ∗ fact 4 ⤳
5 ∗ (4 ∗ fact 3) ⤳
5 ∗ (4 ∗ (3 ∗ fact 2)) ⤳
5 ∗ (4 ∗ (3 ∗ (2 ∗ fact 1))) ⤳
5 ∗ (4 ∗ (3 ∗ (2 ∗ (1 ∗ fact 0)))) ⤳
5 ∗ (4 ∗ (3 ∗ (2 ∗ (1 ∗ 1)))) ⤳ . . . ⤳

The factorial function

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

```
fact 5 ⇝
5 * fact 4 ⇝
5 * (4 * fact 3) ⇝
5 * (4 * (3 * fact 2)) ⇝
5 * (4 * (3 * (2 * fact 1))) ⇝
5 * (4 * (3 * (2 * (1 * fact 0)))) ⇝
5 * (4 * (3 * (2 * (1 * 1)))) ⇝ ... ⇝
120
```

The factorial function

- This product cannot be evaluated!
- It must be kept in memory until the final iteration of **`fact`** has been computed
- For large input values this requires a lot of memory

```
: int =
fact 5
5 * fa                x - 1)
5 * (4
5 * (4
5 * (4 * (3 * (2 * fact 1))) ⇝
5 * (4 * (3 * (2 * (1 * fact 0)))) ⇝
5 * (4 * (3 * (2 * (1 * 1)))) ⇝ ... ⇝
120
```

The factorial function

Time and memory usage are both proportional to the input value. Should we be satisfied by that?

```
fact 1000000;;
```

The factorial function

Time and memory usage are both proportional to the input value. Should we be satisfied by that?

```
fact 1000000;;
```

```
System.StackOverflowException:
    The requested operation caused
    a stack overflow.
```

The factorial function

Time and memory usage are both proportional to the input value. Should we be satisfied by that?

```
fact 1000000;;
System.StackOverflowException:
    The requested operation caused
    a stack overflow.
```

The resulting number is of course too large to fit in an integer, but that is not the problem here

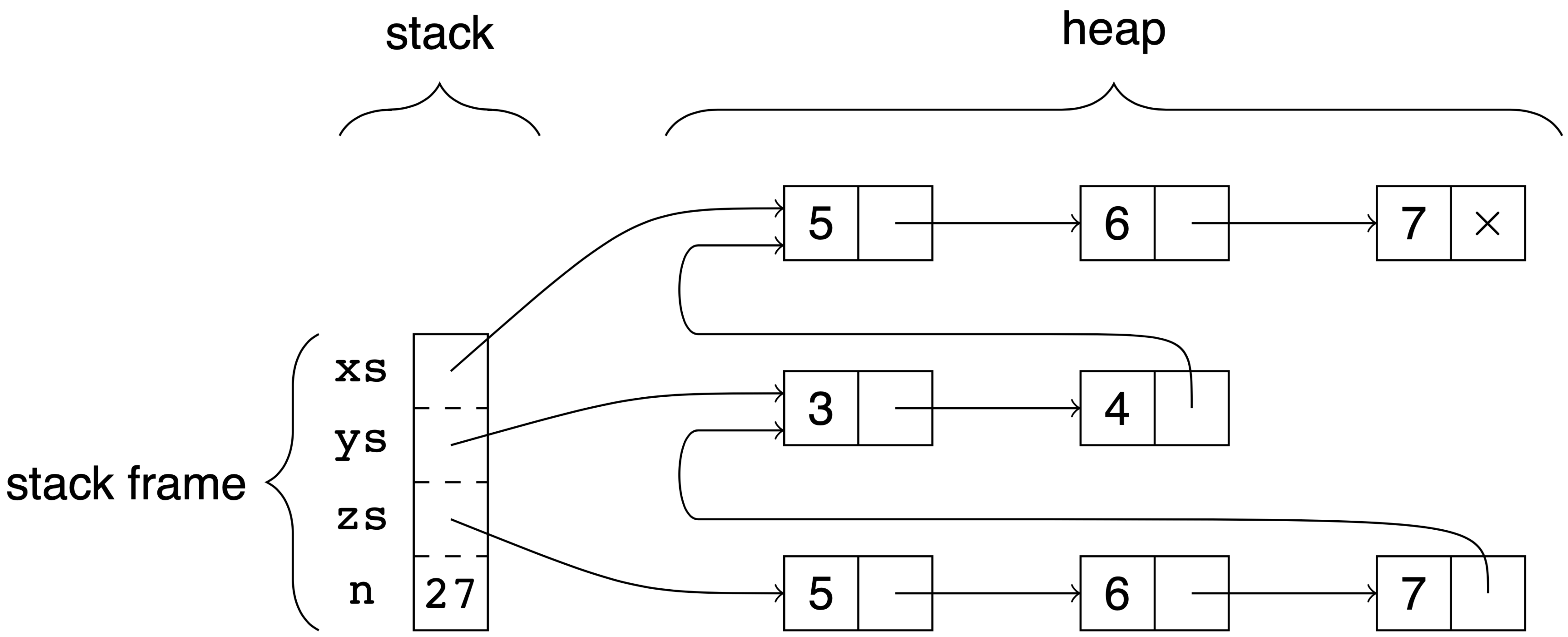
The memory model of F#

To understand what is going on here we need to talk about the **memory model** of F#

The memory model of F#

- Primitive values are allocated on the stack
- Composite values are allocated on the heap

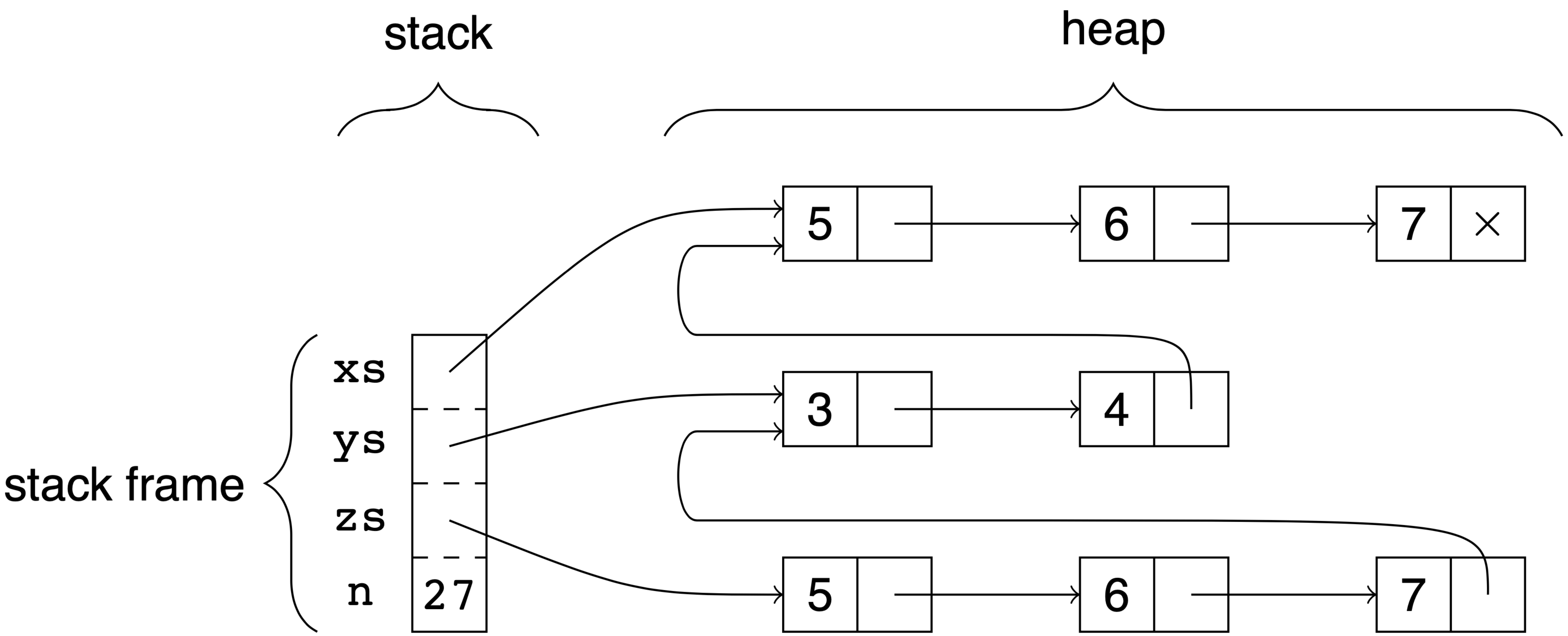
```
let xs = [5; 6; 7]
let ys = 3::4::xs
let zs = xs @ ys
let n = 27
```



The memory model of F#

- The linked list xs is **not copied** when constructing ys
- xs is **only copied** when building xs @ ys

```
let xs = [5; 6; 7]
let ys = 3::4::xs
let zs = xs @ ys
let n = 27
```



Stack frames

```
let zs = let xs = [1; 2]
         let ys = [3; 4]
         xs @ ys
```

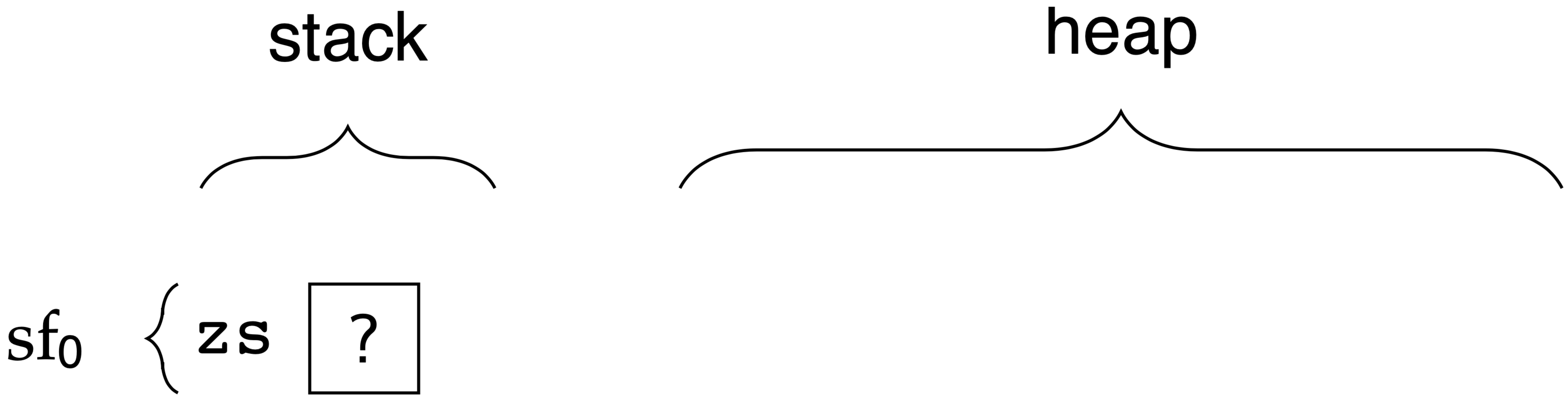
What happens if this let binding of zs is evaluated?

Stack frames

```
let zs = let xs = [1; 2]
         let ys = [3; 4]
         xs @ ys
```

What happens if this let binding of zs is evaluated?

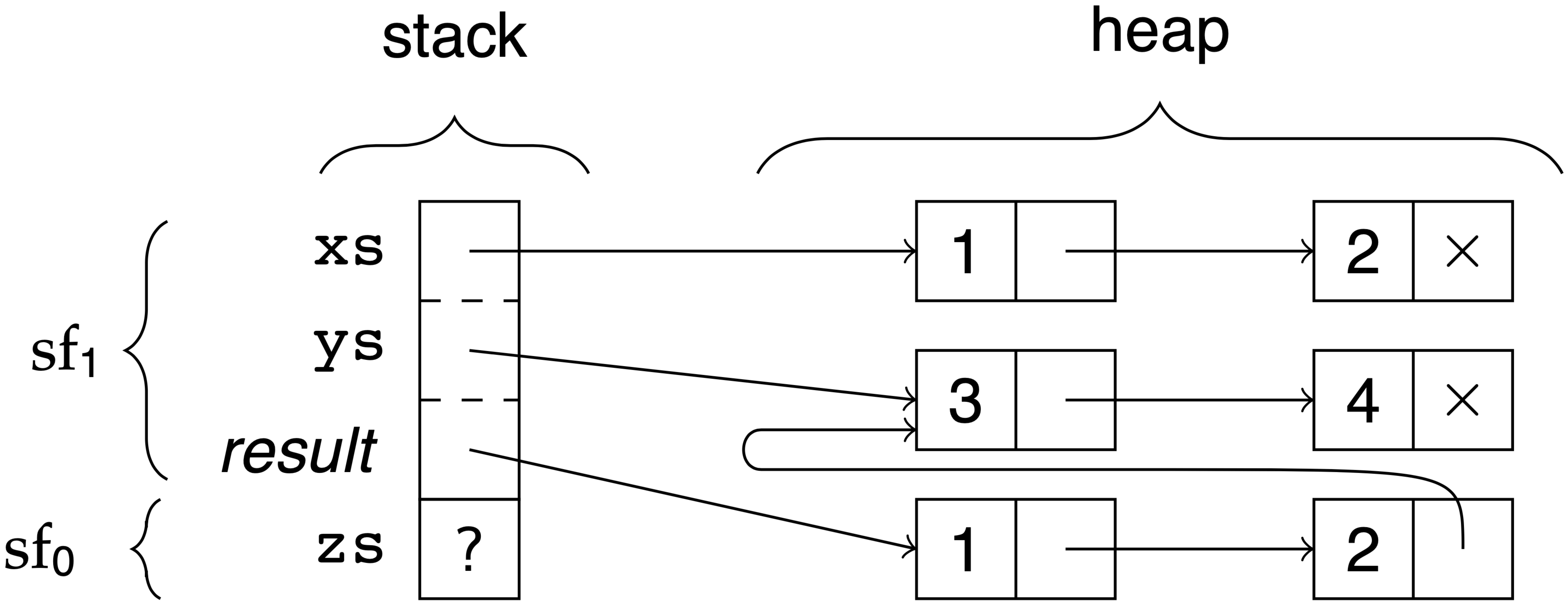
Initial stack and heap prior to the evaluation of the the right-hand side:



Stack frames

```
let zs = let xs = [1; 2]
         let ys = [3; 4]
         xs @ ys
```

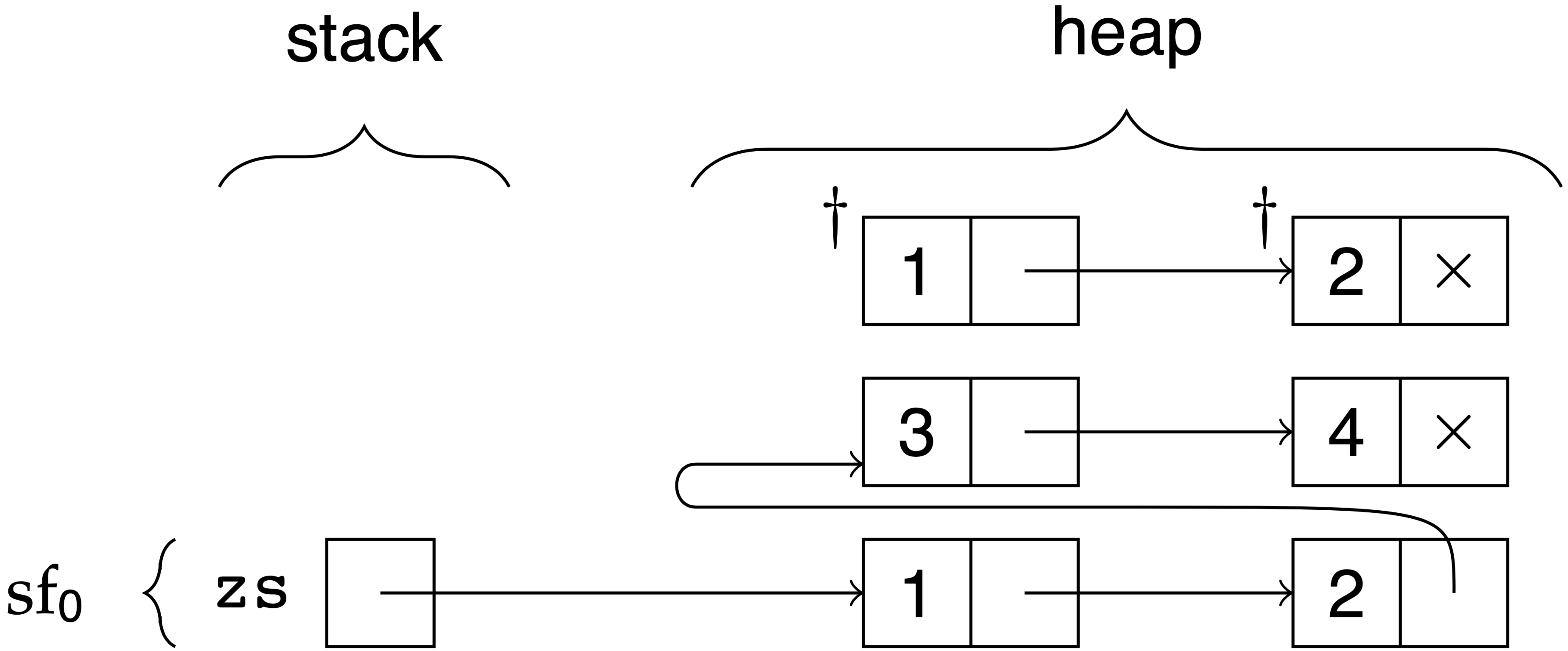
Local declarations are evaluated by pushing a new stack frame onto the stack



Stack frames

```
let zs = let xs = [1; 2]
         let ys = [3; 4]
         xs @ ys
```

The top stack frame is popped when the evaluation of the `let`-expression is done



Stack overflow

- The problems we observed with `fact` have stemmed from the fact that we have **run out of stack space**

- **Each recursive call adds a new stack frame**, which is only released after the recursive call finished

```
fact 5 ⤳
5 * fact 4 ⤳
… ⤳ … ⤳ … ⤳
5 * (4 * (3 * (2 * (1 * fact 0))))
```

Questions?

Part II

Example: List Functions

Two more examples

- We have seen poor memory performance of factorial
- You might not use factorial all that often in practice
- But: many other recursive functions have the same problem (and worse!) if implemented naively

Two more examples

- We have seen poor memory performance of factorial
- You might not use factorial all that often in practice
- But: many other recursive functions have the same problem (and worse!) if implemented naively
- Let’s look at two list functions to illustrate this:
  - List append
  - List reversal

List append

Let’s write our own list append function.
(What can possibly go wrong?)

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []        -> ys
    | x :: xs' -> x :: (app xs' ys)
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (app xs' ys)
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []        -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⇝
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⤳
1 :: (app [2; 3] [4; 5]) ⤳
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⤳
1 :: (app [2; 3] [4; 5]) ⤳
1 :: (2 :: (app [3] [4; 5])) ⤳
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⤳
1 :: (app [2; 3] [4; 5]) ⤳
1 :: (2 :: (app [3] [4; 5])) ⤳
1 :: (2 :: (3 :: (app [] [4; 5]))) ⤳
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []        -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⇝
1 :: (app [2; 3] [4; 5]) ⇝
1 :: (2 :: (app [3] [4; 5])) ⇝
1 :: (2 :: (3 :: (app [] [4; 5]))) ⇝
1 :: (2 :: (3 :: [4; 5])) ⇝
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []          -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⤳
1 :: (app [2; 3] [4; 5]) ⤳
1 :: (2 :: (app [3] [4; 5])) ⤳
1 :: (2 :: (3 :: (app [] [4; 5]))) ⤳
1 :: (2 :: (3 :: [4; 5])) ⤳
1 :: (2 :: [3; 4; 5]) ⤳
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⤳
1 :: (app [2; 3] [4; 5]) ⤳
1 :: (2 :: (app [3] [4; 5])) ⤳
1 :: (2 :: (3 :: (app [] [4; 5]))) ⤳
1 :: (2 :: (3 :: [4; 5])) ⤳
1 :: (2 :: [3; 4; 5]) ⤳
1 :: [2; 3; 4; 5] ⤳
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⤳
1 :: (app [2; 3] [4; 5]) ⤳
1 :: (2 :: (app [3] [4; 5])) ⤳
1 :: (2 :: (3 :: (app [] [4; 5]))) ⤳
1 :: (2 :: (3 :: [4; 5])) ⤳
1 :: (2 :: [3; 4; 5]) ⤳
1 :: [2; 3; 4; 5] ⤳
[1; 2; 3; 4; 5]
```

List append

```
let rec app (xs:'a list) (ys:'a list) : 'a list =
    match xs with
    | []          -> ys
    | x :: xs' -> x :: (app xs' ys)
```

```
app [1; 2; 3] [4; 5] ⇝
1 :: (app [2; 3] [4; 5]) ⇝
1 :: (2 :: (app [3] [4; 5])) ⇝
1 :: (2 :: (3 :: (app [] [4; 5]))) ⇝
1 :: (2 :: (3 :: [4; 5])) ⇝
1 :: (2 :: [3; 4; 5]) ⇝
1 :: [2; 3; 4; 5] ⇝
[1; 2; 3; 4; 5]
```

Time and memory usage are both proportional to the size of `xs`.

Is this satisfactory?

List append

```
> app [1; 2; 3] [1 .. 1000000];;
```

List append

```
> app [1; 2; 3] [1 .. 1000000];;
val it : int list =
[1; 2; 3; 1; 2; 3; 4; 5; …]
```

List append

```
> app [1; 2; 3] [1 .. 1000000];;
val it : int list =
[1; 2; 3; 1; 2; 3; 4; 5; …]

> app [1 .. 1000000] [1; 2; 3];;
```

List append

```
> app [1; 2; 3] [1 .. 1000000];;
val it : int list =
[1; 2; 3; 1; 2; 3; 4; 5; …]

> app [1 .. 1000000] [1; 2; 3];;
System.StackOverflowException:
The requested operation caused
a stack overflow.
```

List append

What about the built-in list append function (`@`) ?
(not the one we just implemented)

List append

What about the built-in list append function (@) ?
(not the one we just implemented)

```
> [1; 2; 3] @ [1..1000000];;
val it : int list =
[1; 2; 3; 1; 2; 3; 4; 5; …]
```

IT UNIVERSITY OF COPENHAGEN

List append

What about the built-in list append function (@) ?
(not the one we just implemented)

```
> [1; 2; 3] @ [1..1000000];;
val it : int list =
[1; 2; 3; 1; 2; 3; 4; 5; …]
```

```
> [1..1000000] @ [1; 2; 3];;
val it : int list =
[1; 2; 3; 4; 5; 6; 7; …]
```

List append

What about the built-in list append function (@) ?
(not the one we just implemented)

```
> [1; 2; 3] @ [1..1000000];;
val it : int list =
[1; 2; 3; 1; 2; 3; 4; 5; …]
```

```
> [1..1000000] @ [1; 2; 3];;
val it : int list =
[1; 2; 3; 4; 5; 6; 7; …]
```

So what is going on here?

List append

What about the built-in list append function (@) ?
(not the one we just implemented)

```
> [1; 2; 3] @ [1..1000000];;
val it : int list =
[1; 2; 3; 1; 2; 3; 4; 5; …]
```

So what is going on here?

```
> [1..1000000] @ [1; 2; 3];;
val it : int list =
[1; 2; 3; 4; 5; 6; 7; …]
```

What did we get wrong?

List reversal

Before looking at how to solve this problem, let's consider another example!

List reversal

Before looking at how to solve this problem, let's consider another example!

A naive version of list reversal can be written as follows:

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []          -> []
    | x :: xs' -> rev xs' @ [x]
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []          -> []
    | x :: xs' -> rev xs' @ [x]
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []         -> []
    | x :: xs' -> rev xs' @ [x]
```

```
rev [1; 2; 3; 4] ⇝
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []          -> []
    | x :: xs' -> rev xs' @ [x]
```

```
rev [1; 2; 3; 4] ⇝
(rev [2; 3; 4]) @ [1] ⇝
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []          -> []
    | x :: xs' -> rev xs' @ [x]
```

```
rev [1; 2; 3; 4] ⇝
(rev [2; 3; 4]) @ [1] ⇝
((rev [3; 4]) @ [2]) @ [1] ⇝
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []         -> []
    | x :: xs' -> rev xs' @ [x]
```

```
rev [1; 2; 3; 4] ⤳
(rev [2; 3; 4]) @ [1] ⤳
((rev [3; 4]) @ [2]) @ [1] ⤳
(((rev [4]) @ [3]) @ [2]) @ [1] ⤳
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []         -> []
    | x :: xs' -> rev xs' @ [x]
```

```
rev [1; 2; 3; 4] ⇝
(rev [2; 3; 4]) @ [1] ⇝
((rev [3; 4]) @ [2]) @ [1] ⇝
(((rev [4]) @ [3]) @ [2]) @ [1] ⇝
((((rev []) @ [4]) @ [3]) @ [2]) @ [1] ⇝
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []          -> []
    | x :: xs' -> rev xs' @ [x]
```

```
rev [1; 2; 3; 4] ⤳
(rev [2; 3; 4]) @ [1] ⤳
((rev [3; 4]) @ [2]) @ [1] ⤳
(((rev [4]) @ [3]) @ [2]) @ [1] ⤳
((((rev []) @ [4]) @ [3]) @ [2]) @ [1] ⤳
((([] @ [4]) @ [3]) @ [2]) @ [1] ⤳ … ⤳
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []         -> []
    | x :: xs' -> rev xs' @ [x]
```

```
rev [1; 2; 3; 4] ⤳
(rev [2; 3; 4]) @ [1] ⤳
((rev [3; 4]) @ [2]) @ [1] ⤳
(((rev [4]) @ [3]) @ [2]) @ [1] ⤳
((((rev []) @ [4]) @ [3]) @ [2]) @ [1] ⤳
((([] @ [4]) @ [3]) @ [2]) @ [1] ⤳ … ⤳
[4; 3; 2; 1]
```

List reversal

```
rev [1; 2; 3; 4] ⇝
(rev [2; 3; 4]) @ [1] ⇝
((rev [3; 4]) @ [2]) @ [1] ⇝
(((rev [4]) @ [3]) @ [2]) @ [1] ⇝
((((rev []) @ [4]) @ [3]) @ [2]) @ [1] ⇝
((([] @ [4]) @ [3]) @ [2]) @ [1] ⇝ … ⇝
[4; 3; 2; 1]
```

- Memory requirements?
- Computation time?

List reversal

```
> rev [1..1000000];;
System.StackOverflowException:
The requested operation caused
a stack overflow.
```

IT UNIVERSITY OF COPENHAGEN

Summary

- Recursive functions use the **stack** to remember where they are in the recursion
- This can lead to a **stack overflow**
- Naive recursive functions may have **asymptotically poor performance**, e.g. list reversal $O(n^2)$ runtime

Questions?

Part IIIA

Iterative Functions

(aka. tail recursion)

Iterative functions

- **Iterative functions** have all recursive calls as their last operation ($\Rightarrow$ also called **tail recursion**)

- F# will optimise iterative functions so that they don't increase the stack

$\Rightarrow$ **No stack overflows!**

- <u>Two approaches</u>: We can use **accumulators** or **continuations** to make recursive functions iterative

Accumulators

Accumulators store the value that has been computed so far

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Original factorial function from earlier

Accumulators

Accumulators store the value that has been computed so far

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Original factorial function from earlier

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

factorial with an accumulator

Accumulators

Accumulators store the value that has been computed so far

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

Original factorial function from earlier

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

recursion is the last action!

Accumulators

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)

let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

Accumulators

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

factA satisfies this property:

```
  factA acc x
= acc * fact x
```

Accumulators

```
let rec fact (x : int) : int =
    match x with
    | 0 -> 1
    | x -> x * fact (x - 1)
```

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

factA satisfies this property:

```
  factA acc x
= acc * fact x
```

In particular,
if acc = 1

```
  factA 1 x
= 1 * fact x
= fact x
```

Accumulators

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

factA 1 5 ⤳ factA (1 * 5) (5 - 1) ⤳

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

```
factA 1 5   ⤳ factA (1 * 5) (5 - 1)   ⤳
factA 5 4   ⤳ factA (5 * 4) (4 - 1)   ⤳
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

```
factA 1 5   ⇝ factA (1 * 5) (5 - 1)   ⇝
factA 5 4   ⇝ factA (5 * 4) (4 - 1)   ⇝
factA 20 3  ⇝ factA (20 * 3) (3 - 1)  ⇝
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

```
factA 1 5    ⇝ factA (1 * 5) (5 - 1)    ⇝
factA 5 4    ⇝ factA (5 * 4) (4 - 1)    ⇝
factA 20 3   ⇝ factA (20 * 3) (3 - 1)   ⇝
factA 60 2   ⇝ factA (60 * 2) (2 - 1)   ⇝
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

```
factA 1 5   ⇝ factA (1 * 5) (5 - 1)     ⇝
factA 5 4   ⇝ factA (5 * 4) (4 - 1)     ⇝
factA 20 3  ⇝ factA (20 * 3) (3 - 1)   ⇝
factA 60 2  ⇝ factA (60 * 2) (2 - 1)   ⇝
factA 120 1 ⇝ factA (120 * 1) (1 - 1) ⇝
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

```
factA 1 5   ⇝ factA (1 * 5) (5 - 1)     ⇝
factA 5 4   ⇝ factA (5 * 4) (4 - 1)     ⇝
factA 20 3  ⇝ factA (20 * 3) (3 - 1)   ⇝
factA 60 2  ⇝ factA (60 * 2) (2 - 1)   ⇝
factA 120 1 ⇝ factA (120 * 1) (1 - 1) ⇝
factA 120 0 ⇝ 120
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

Accumulators

```
let rec factA (acc : int) (x : int) : int =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
```

```
fact 5 ⤳ 5 * fact 4 ⤳
5 * (4 * fact 3) ⤳
5 * (4 * (3 * fact 2)) ⤳
5 * (4 * (3 * (2 * fact 1))) ⤳
5 * (4 * (3 * (2 * (1 * fact 0)))) ⤳
5 * (4 * (3 * (2 * (1 * 1)))) ⤳ … ⤳ 120
```

Accumulators

let

fact needs to 'remember' remaining multiplications

factA already performed the multiplications on the accumulator

```
fact 5 ⤳ 5 * fact 4 ⤳
5 * (4 * fact 3) ⤳
5 * (4 * (3 * fact 2)) ⤳
5 * (4 * (3 * (2 * fact 1))) ⤳
5 * (4 * (3 * (2 * (1 * fact 0)))) ⤳
5 * (4 * (3 * (2 * (1 * 1)))) ⤳ … ⤳ 120
```

Accumulators

fact needs to 'remember' remaining multiplications

factA already performed the multiplications on the accumulator

Instead of

```
5 * (4 * (3 * (2 * (1 * 1))))
```

we compute

```
((((1 * 5) * 4) * 3) * 2) * 1
```

Accumulators

```
> fact 1000000;;
    System.StackOverflowException:
    The requested operation caused
    a stack overflow.

> factA 1 1000000;;
    val it : int = 0
```

The result is clearly incorrect (the number is stupendously large) but it does not overflow the stack

Accumulators

```
> fact 1000000;;
    System.StackOverflowException:
    The requested operation caused
    a stack overflow.
```

```
> factA 1 1000000;;
    val it : int = 0
```

The result is clearly incorrect (the number is stupendously large) but it does not overflow the stack

Time: linear
Space: constant

Putting a bow on it

```
let rec factA acc x =
  match x with
  | 0 -> acc
  | x -> factA (acc * x) (x - 1)
```

Putting a bow on it

```
let rec factA acc x =
  match x with
  | 0 -> acc
  | x -> factA (acc * x) (x - 1)
```

This function exposes implementation details:

```
factA : int -> int -> int
```

Putting a bow on it

```
let rec factA acc x =
  match x with
  | 0 -> acc
  | x -> factA (acc * x) (x - 1)
```

This function exposes implementation details:

```
factA : int -> int -> int
```

```
let fact x =
  let rec factA acc x =
    match x with
    | 0 -> acc
    | x -> factA (acc * x) (x - 1)
  factA 1 x
```

We can avoid this by nesting function definitions:

```
fact : int -> int
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []         -> []
    | x :: xs' -> rev xs' @ [x]
```

List reversal

```
let rec rev (xs : 'a list) : 'a list =
    match xs with
    | []        -> []
    | x :: xs' -> rev xs' @ [x]

let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []        -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

List reversal

Time: linear
Space: linear

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []        -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

List reversal

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []        -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l

revA [1; 2; 3; 4] ⇝
```

IT UNIVERSITY OF COPENHAGEN

List reversal

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []         -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

revA [1; 2; 3; 4] ⇝ aux [] [1; 2; 3; 4] ⇝

List reversal

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []       -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

revA [1; 2; 3; 4] ⇝ aux [] [1; 2; 3; 4] ⇝
aux [1] [2; 3; 4] ⇝

List reversal

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []        -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

```
revA [1; 2; 3; 4] ⇝ aux [] [1; 2; 3; 4] ⇝
aux [1] [2; 3; 4] ⇝ aux [2; 1] [3; 4]   ⇝
```

List reversal

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []        -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

```
revA [1; 2; 3; 4] ⇝ aux [] [1; 2; 3; 4] ⇝
aux [1] [2; 3; 4] ⇝ aux [2; 1] [3; 4]    ⇝
aux [3; 2; 1] [4] ⇝
```

List reversal

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []        -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

```
revA [1; 2; 3; 4] ⇝ aux [] [1; 2; 3; 4] ⇝
aux [1] [2; 3; 4] ⇝ aux [2; 1] [3; 4]   ⇝
aux [3; 2; 1] [4] ⇝ aux [4; 3; 2; 1] [] ⇝
```

List reversal

```
let revA (l : 'a list) : 'a list =
    let rec aux acc xs =
        match xs with
        | []        -> acc
        | x :: xs' -> aux (x :: acc) xs'
    aux [] l
```

revA [1; 2; 3; 4] ⤳ aux [] [1; 2; 3; 4] ⤳
aux [1] [2; 3; 4] ⤳ aux [2; 1] [3; 4] ⤳
aux [3; 2; 1] [4] ⤳ aux [4; 3; 2; 1] [] ⤳
[4; 3; 2; 1]

Measuring time & memory usage

The **#time** command enables time and memory measurements in the interactive environment

```
> #time;;
--> Timing now on
```

Measuring time & memory usage

The **#time** command enables time and ~~memory~~ GC measurements in the interactive environment

```
> #time;;
--> Timing now on
```

Measuring time & memory usage

The **#time** command enables time and ~~memory~~ GC measurements in the interactive environment

```
> #time;;
--> Timing now on

> rev [1..20000];;
Real: 00:00:17.506,
CPU:0:00:16.540,
GC gen0: 794, gen1: 0
val it : int list =
 [20000; 19999; 19998;…]
```

The garbage collector

- The garbage collector reclaims unused memory
- The heap is divided into generations: gen0, gen1, gen2
- gen0 is the youngest and gen2 is the oldest
- Data typically dies young and the GC is designed to take advantage of that
- The `#time` command counts the number of GC passes that were performed

Measuring time & memory usage

```
> rev [1..20000];;
Real: 00:00:17.506,
CPU:0:00:16.540,
GC gen0: 794, gen1: 0
val it : int list =
  [20000; 19999; 19998;…]
```

Naively reversing 20000 elements takes around 17 seconds and 794 garbage collections

Measuring time & memory usage

```
> rev [1..20000];;
Real: 00:00:17.506,
CPU:0:00:16.540,
GC gen0: 794, gen1: 0
val it : int list =
  [20000; 19999; 19998;…]
```

Naively reversing 20000 elements takes around 17 seconds and 794 garbage collections

```
> revA [1..20000];;
Real: 00:00:00.001,
CPU: 00:00:00.001,
GC gen0: 0, gen1: 0
val it : int list =
  [20000; 19999; 19998;…]
```

Reversing 20000 elements using an accumulator takes around 1 millisecond and no garbage collections

Observation

- We can use an **accumulator** if we can can compute the same result in a **different order**
- For example, fact 5 computes

```
5 * (4 * (3 * (2 * (1 * 1))))
```

- whereas factA 5 computes

```
((((1 * 5) * 4) * 3) * 2) * 1
```

- Multiplication is associative!

Example: rev

- Similarly, `rev [1;2;3;4]` computes

```
((([] @ [4]) @ [3]) @ [2]) @ [1]
```

- whereas `revA [1;2;3;4]` computes

```
[4] @ ([3] @ ([2] @ ([1] @ [])))
```

- which is equal to

```
4 :: (3 :: (2 :: (1 :: [])))
```

- `@` is associative!

Take-home message

The stack is large, but the heap is larger. Get into a habit of writing iterative functions!

Questions?

Part IIIB

Continuations

When accumulators fail

- Accumulators are great when they work
- But they do **not work every time**, e.g. when
  - we **cannot reorder** the way a function computes its results (e.g. the `foldBack` function)
  - we have **multiple recursive calls** that cannot be combined into one (e.g. tree traversal)

Append

```
let rec append xs ys =
    match xs with
    | []       -> ys
    | x :: xs' -> x :: (append xs' ys)
```

Append

```
let rec append xs ys =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (append xs' ys)
```

not a tail call

Append

```
let rec append xs ys =
    match xs with
    | []        -> ys
    | x :: xs' -> x :: (append xs' ys)
```

not a tail call

- An **accumulator will not work** (not directly at least).
- Instead, we use a **continuation** to get tail recursion.
- A continuation is a function that is meant **to be called after** the recursive function is finished

Append

```
let rec append xs ys =
    match xs with
    | []       -> ys
    | x :: xs' -> x :: (append xs' ys)
```

not a tail call

Idea: make the recursive call, but remember that x still has to be added afterwards

Goal: write a recursive function appendC that takes an additional argument `c : ‘a list -> ‘a list` such that

```
appendC xs ys c = c (append xs ys)
```

IT UNIVERSITY OF COPENHAGEN

Append

```
let rec append xs ys =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (append xs' ys)


let rec appendC xs ys c =
    match xs with
    | []          -> c ys
    | x :: xs' -> appendC xs' ys
                          (fun r -> c (x :: r))
```

Append

```
let rec append xs ys =
    match xs with
    | []         -> ys
    | x :: xs' -> x :: (append xs' ys)
```

```
let rec appendC xs ys c =
    match xs with
    | []           -> c ys
    | x :: xs' -> appendC xs' ys
                          (fun r -> c (x :: r))
```

After the recursion is complete: add x and then apply the continuation

Append

```
let rec appendC xs ys c =
    match xs with
    | []          -> c ys
    | x :: xs' -> appendC xs' ys
                               (fun r -> c (x :: r))
```

```
let (@) xs ys = appendC xs ys id
```

This definition works because of the following property:

```
appendC xs ys c = c (append xs ys)
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []         -> c ys
    | x :: xs' -> appendC xs' ys
                           (fun r -> c (x :: r))
```

```
let (@) xs ys = appendC xs ys id
```

```
id = fun x -> x
```

This definition works because of the following property:

```
appendC xs ys c = c (append xs ys)
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []        -> c ys
    | x :: xs' -> appendC xs' ys
                         (fun r -> c (x :: r))
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []         -> c ys
    | x :: xs' -> appendC xs' ys
                          (fun r -> c (x :: r))
```

```
[1;2] @ [3]
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []         -> c ys
    | x :: xs' -> appendC xs' ys
                       (fun r -> c (x :: r))
```

```
[1;2] @ [3] ⇝ appendC [1;2] [3] id ⇝
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []        -> c ys
    | x :: xs' -> appendC xs' ys
                      (fun r -> c (x :: r))
```

[1;2] @ [3] ⇝ appendC [1;2] [3] id ⇝

appendC [2] [3] (fun r -> id (1::r)) ⇝

Append

```
let rec appendC xs ys c =
    match xs with
    | []         -> c ys
    | x :: xs' -> appendC xs' ys
                               (fun r -> c (x :: r))
```

```
[1;2] @ [3] ⇝ appendC [1;2] [3] id ⇝
appendC [2] [3] (fun r -> id (1::r)) ⇝
appendC [] [3] (fun r -> (fun r -> id (1::r)) (2::r)) ⇝
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []          -> c ys
    | x :: xs' -> appendC xs' ys
                            (fun r -> c (x :: r))
```

```
[1;2] @ [3] ⤳ appendC [1;2] [3] id ⤳
appendC [2] [3] (fun r -> id (1::r)) ⤳
appendC [] [3] (fun r -> (fun r -> id (1::r)) (2::r)) ⤳
(fun r -> (fun r -> id (1::r)) (2::r)) [3] ⤳
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []          -> c ys
    | x :: xs' -> appendC xs' ys
                                (fun r -> c (x :: r))
```

```
[1;2] @ [3] ⤳ appendC [1;2] [3] id ⤳
appendC [2] [3] (fun r -> id (1::r)) ⤳
appendC [] [3] (fun r -> (fun r -> id (1::r)) (2::r)) ⤳
(fun r -> (fun r -> id (1::r)) (2::r)) [3] ⤳
(fun r -> id (1::r)) [2;3]
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []         -> c ys
    | x :: xs' -> appendC xs' ys
                          (fun r -> c (x :: r))
```

```
[1;2] @ [3] ⤳ appendC [1;2] [3] id ⤳
appendC [2] [3] (fun r -> id (1::r)) ⤳
appendC [] [3] (fun r -> (fun r -> id (1::r)) (2::r)) ⤳
(fun r -> (fun r -> id (1::r)) (2::r)) [3] ⤳
(fun r -> id (1::r)) [2;3] ⤳ id [1;2;3]
```

Append

```
let rec appendC xs ys c =
    match xs with
    | []          -> c ys
    | x :: xs' -> appendC xs' ys
                          (fun r -> c (x :: r))
```

```
[1;2] @ [3] ⤳ appendC [1;2] [3] id ⤳
appendC [2] [3] (fun r -> id (1::r)) ⤳
appendC [] [3] (fun r -> (fun r -> id (1::r)) (2::r)) ⤳
(fun r -> (fun r -> id (1::r)) (2::r)) [3] ⤳
(fun r -> id (1::r)) [2;3] ⤳ id [1;2;3] ⤳ [1;2;3]
```

When accumulators fail

- Recall accumulators fail if
  - we **cannot reorder** the way a function computes its results.
  - we have **multiple recursive calls** that cannot be combined into one.

When accumulators fail

- Recall accumulators fail if
  - we **cannot reorder** the way a function computes its results.
  - we have **multiple recursive calls** that cannot be combined into one.

List append needs to add elements in the right order

When accumulators fail

- Recall accumulators fail if
  - we **cannot reorder** the way a function computes its results.
  - we have **multiple recursive calls** that cannot be combined into one.

List append needs to add elements in the right order

Next example:

Tree traversal

Example: Tree traversal

```
type BinTree =
    | Leaf
    | Node of BinTree * int * BinTree

let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n
```

Example: Tree traversal

```
type BinTree =
    | Leaf
    | Node of BinTree * int * BinTree

let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let t = genTree 1000000
printfn "%d" (sum t)
```

generates a tree of height 1,000,000

Example: Tree traversal

```
type BinTree =
    | Leaf
    | Node of BinTree * int * BinTree

let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let t = genTree 1000000
printfn "%d" (sum t)
```

generates a tree of height 1,000,000

⇒ Stack overflow

Let’s Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let rec sumA (t : BinTree) (acc : int) : int =
    match t with
    | Leaf -> acc
    | Node (l,n,r) -> sumA r (sumA l (n + acc))
```

Let’s Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let rec sumA (t : BinTree) (acc : int) : int =
    match t with
    | Leaf -> acc
    | Node (l,n,r) -> sumA r (sumA l (n + acc))
```

not a tail call

Let's Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n

let rec sumA (t : BinTree) (acc : int) : int =
    match t with
    | Leaf -> acc
    | Node (l,n,r) -> sumA r (sumA l (n + acc))
```

not a tail call

We need a way to say "compute sum of `l` and afterwards **continue** with `r`" while only using one recursive call

Let's Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n
```

Goal: write a recursive function

```
sumC : BinTree -> (int -> int) -> int
```

such that

```
sumC t c = c (sum t)
```

Hence: `sum t = sumC t id`

Let’s Try an Accumulator

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n
```

Goal: write a recursive function

```
sumC : BinTree -> (int -> int) -> int
```

such that

```
sumC t c = c (sum t)
```

Hence: `sum t = sumC t id`

sum with a continuation

```
let rec sum (t : BinTree) : int =
    match t with
    | Leaf -> 0
    | Node (l,n,r) -> sum l + sum r + n


let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
sumC (Node (Leaf, 4, Leaf)) id
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
   sumC (Node (Leaf, 4, Leaf)) id
⤳ sumC Leaf (fun vl -> sumC Leaf
                 (fun vr -> id (vl + vr + 4)))
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
  sumC (Node (Leaf, 4, Leaf)) id
⤳ sumC Leaf (fun vl -> sumC Leaf
               (fun vr -> id (vl + vr + 4)))
⤳ sumC Leaf (fun vr -> id (0 + vr + 4))
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
  sumC (Node (Leaf, 4, Leaf)) id
⤳ sumC Leaf (fun vl -> sumC Leaf
                (fun vr -> id (vl + vr + 4)))
⤳ sumC Leaf (fun vr -> id (0 + vr + 4))
⤳ id (0 + 0 + 4)
```

sum with a continuation

```
let rec sumC (t : BinTree) (c : int -> int) : int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) ->
    sumC l (fun vl ->
           sumC r (fun vr -> c (vl + vr + n)))
```

```
  sumC (Node (Leaf, 4, Leaf)) id
⤳ sumC Leaf (fun vl -> sumC Leaf
             (fun vr -> id (vl + vr + 4)))
⤳ sumC Leaf (fun vr -> id (0 + vr + 4))
⤳ id (0 + 0 + 4)   ⤳ 4
```

The type of continuations

```
let rec sumC t (c : int -> int) :int =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) -> sumC l (fun vl ->
              sumC r (fun vr -> c(vl + vr + n)))
```

If you don’t provide type annotations F# will infer a more general type for the continuations.

The type of continuations

```
let rec sumC t (c : int -> 'a ) :'a  =
  match t with
   | Leaf -> c 0
   | Node (l,n,r) -> sumC l (fun vl ->
                sumC r (fun vr -> c(vl + vr + n)))
```

If you don't provide type annotations F# will infer a more general type for the continuations.

The type of continuations

```
let rec sumC t (c : int -> ‘a ) :‘a  =
  match t with
  | Leaf -> c 0
  | Node (l,n,r) -> sumC l (fun vl ->
               sumC r (fun vr -> c(vl + vr + n)))
```

If you don’t provide type annotations F# will infer a more general type for the continuations.

This general type is beneficial: If you forget to call the continuation or recursion in a non-tail position, you will get a type error!

Why does this work?

With **continuation** we can **store computations** to be executed in the future.

Given a function `f : A -> B`

we write a function `fC : A -> (B -> B) -> B` so that

$$f_C\ v_0\ c_0 \rightsquigarrow f_C\ v_1\ c_1 \rightsquigarrow \ldots \rightsquigarrow f_C\ v_n\ c_n \rightsquigarrow c_n\ (f\ v_n)$$

and $c_k\ (f\ v_k) = f\ v_0$ for all k

Summary

- Recursive functions use the **stack** to remember where they are in the recursion
- This can lead to a **stack overflow**
- Solution: transform recursive functions into **iterative functions**, where recursive calls are always last
- Two approaches: **accumulator** (does not always work) & **continuation** (works always)
- This can also lead to **algorithmic improvements** (e.g. in the list reversal function)

IT UNIVERSITY OF COPENHAGEN

Questions?