Designing Functions (OCaml Practice)

Concepts of Programming Languages
Lecture 3

CAS CS 320

where we left off...

Recall: What is a list?

```
let _ = 1 :: 2 :: 3 :: []
let _ = 1 :: (2 :: (3 :: []))
let _ = [1; 2; 3]
```

A list is an ordered ***variable-length*** ***homogeneous*** collection of data

Many important operations on data can be represented as operations on lists (e.g., updating all users in a database)

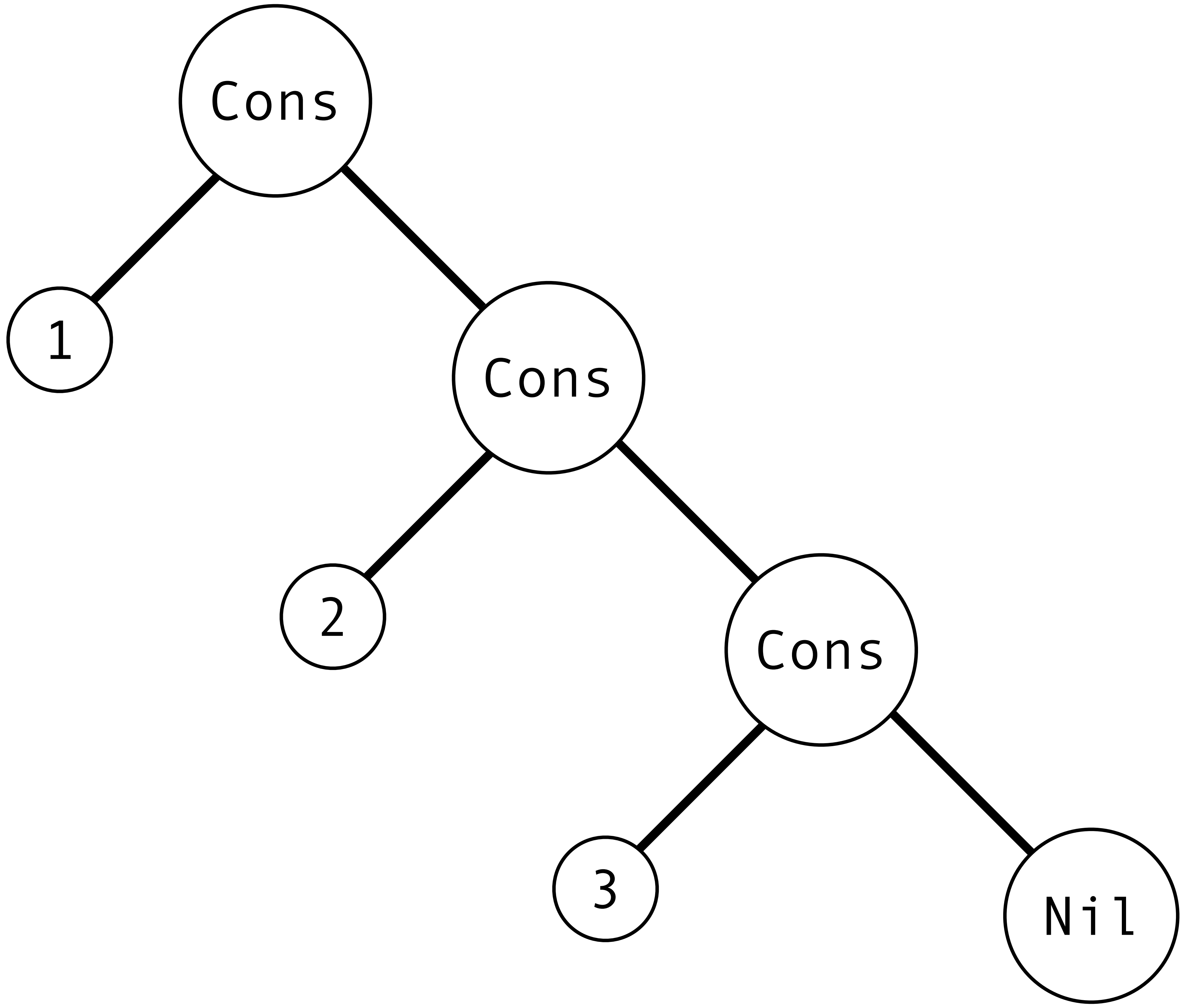
Recall: The Picture

We can think of the list

`1 :: 2 :: 3 :: []`

as a leaning tree with data a leaves

(this will generalize to other *algebraic* data types)



A Note on Polymorphism

```
let rec length l =
  match l with
  | [] -> 0
  | x :: xs -> 1 + length xs
```

What is the type of the length function?

Does this function depend on the values in the list?

The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	`string list`	`int list list`

The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	**`string list`**	**`int list list`**

The list type is an example of a **parametrized** type

The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	**`string list`**	**`int list list`**

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	**`string list`**	**`int list list`**

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

For this, we need *type parameters* to stand for *any* type:

The List Type

`[1;2;3]` — **int list**

`["1";"2";"3"]` — **string list**

`[[1;1];[2;2];[3;3]]` — **int list list**

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

For this, we need *type parameters* to stand for *any* type:

'a, 'b, 'c,...

Not all functions can be polymorphic

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

Not all functions can be polymorphic

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

Can this function be applied to a list parametrized by any type?

Not all functions can be polymorphic

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

Can this function be applied to a list parametrized by any type?

Answer: No, it can only be applied to **int lists**

Not all functions can be polymorphic

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

Can this function be applied to a list parametrized by any type?

Answer: No, it can only be applied to **int lists**

OCaml's type inference is good at "guessing" when functions are polymorphic

Tail Recursion

demo

(even the wrong way)

Tail Recursion

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

not tail recursive

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

tail recursive

A recursive function is **tail recursive** if it does not perform any computations on the result of a recursive call

Why do we care?

Why do we care?

Recursive functions are *expensive* with respect to the call-stack

Why do we care?

Recursive functions are *expensive* with respect to the call-stack

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames

Why do we care?

Recursive functions are *expensive* with respect to the call-stack

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames

Tail-recursive functions "behave iteratively"

The Picture

```
fact 5
```

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

The Picture

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

fact 5

fact 4

The Picture

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

fact 5

fact 4

fact 3

The Picture

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

fact 5

fact 4

fact 3

fact 2

The Picture

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

fact 5

fact 4

fact 3

fact 2

fact 1

The Picture

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

fact 5

fact 4

fact 3

fact 2

fact 1

fact 0

The Picture

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

```
fact 5

fact 4

fact 3

fact 2

fact 1

fact 0
```

$\Longrightarrow$ **1**

The Picture

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

fact 5

fact 4

fact 3

fact 2

fact 1

$\Longrightarrow$ **1 * 1 = 1**

The Picture

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

```
fact 5
```

```
fact 4
```

```
fact 3
```

```
fact 2

⟹ 2 * 1 = 2
```

The Picture

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

```
fact 5
```

```
fact 4
```

```
fact 3
⟹ 3 * 2 = 6
```

The Picture

```
fact 5
```

```
fact 4
⟹ 4 * 6 = 24
```

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

The Picture

```
fact 5

⟹ 5 * 24 = 120
```

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

The Picture

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

fact 5

⟹ **5 * 24 = 120**

fact 4

⟹ **4 * 6 = 24**

fact 3

⟹ **3 * 2 = 6**

fact 2

⟹ **2 * 1 = 2**

fact 1

⟹ **1 * 1 = 1**

fact 0

⟹ **1**

1 frame per recursive call

The Picture

```
loop 1 5
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
loop 1 5
```

```
loop 5 4
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

```
fact 60 2
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

```
fact 60 2
```

```
fact 120 1
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

```
fact 60 2
```

```
fact 120 1
```

```
fact 120 0
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

```
fact 60 2
```

```
fact 120 1
```

```
fact 120 0

⟹ 120
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2

fact 120 1

$\Longrightarrow$ **120**

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

```
fact 60 2
⟹ 120
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
⟹ 120
```

The Picture

```
loop 1 5
```

```
loop 5 4
⟹ 120
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
loop 1 5
⟹ 120
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

loop 1 5

$\Longrightarrow$ **120**

loop 5 4

$\Longrightarrow$ **120**

loop 20 3

$\Longrightarrow$ **120**

fact 60 2

$\Longrightarrow$ **120**

fact 120 1

$\Longrightarrow$ **120**

fact 120 0

$\Longrightarrow$ **120**

1 frame per recursive call

BUT THE VALUE DOESN'T CHANGE ON IT'S WAY UP THE CALL STACK

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 5 4
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 20 3
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 120 1
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 120 0
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 120 0
```

$\Longrightarrow$ **120**

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

loop 120 0
$\Longrightarrow$ **120**

1 frame *for every* recursive call

Tail Position

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

computation after the recursive call

not tail recursive

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

tail recursive

Tail-call optimizations apply to functions whose recursive calls are in **tail position**

Intuition: A call is in tail position if there is no computation *after* the recursive call

Aside: Tail Position More Formally

```
let rec f x1 x2 ... xk = e
```

A recursive call **f e1 e2 ... ek** is in tail position in **e** if:

- it does not appear in **e**, or **e** is the recursive call itself
- e = **if e1 then e2 else e3** and the call does not appear in **e1** and it is in tail position in **e2** and **e3**
- e is a **match-expression** and the call is in tail position in every branch, and does not appear in the matched expression
- e = **let x = e1 in e2** and the call does not appear in the **e1** and it is in tail position in **e2**

Aside: Tail Position More Formally

```
let rec f x1 x2 ... xk = e
```

A recursive call **f e1 e2 ... ek*** is in tail position in **e** if:

- » it does not appear in **e**, or **e** is the recursive call itself
- » e = **if e1 then e2 else e3** and the call does not appear in **e1** and it is in tail position in **e2** and **e3**
- » e is a **match-expression** and the call is in tail position in every branch, and does not appear in the matched expression
- » e = **let x = e1 in e2** and the call does not appear in the **e1** and it is in tail position in **e2**

* f cannot appear in e1 ... ek

Tail Recursion and Lists

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

Tail Recursion and Lists

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

We need to take care with tail-recursion and lists

Tail Recursion and Lists

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

We need to take care with tail-recursion and lists

Does the above program concatenate two lists?

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
append [1;2;3] [4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [1;2;3] [4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match [1;2;3] with
| [] -> [4;5;6]
| x :: xs -> loop xs (x :: [4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match 1 :: [2;3] with
| [] -> [4;5;6]
| x :: xs -> loop xs (x :: [4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [2;3] (1 :: [4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [2;3] [1;4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match [2;3] with
| [] -> [1;4;5;6]
| x :: xs -> loop xs (x :: [1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match 2 :: [3] with
| [] -> [1;4;5;6]
| x :: xs -> loop xs (x :: [1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [3] (2 :: [1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [3] [2;1;4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match [3] with
| [] -> [2;1;4;5;6]
| x :: xs -> loop xs (x :: [2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match 3 :: [] with
| [] -> [2;1;4;5;6]
| x :: xs -> loop xs (x :: [2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [] (3 :: [2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [] [3;2;1;4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match [] with
| [] -> [3;2;1;4;5;6]
| x :: xs -> loop xs (x :: [3;2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

[3;2;1;4;5;6]

whoops!

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

We need to take care with tail-recursion and lists

Does the above program concatenate two lists?

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

should be (List.rev l)

We need to take care with tail-recursion and lists

Does the above program concatenate two lists?

Accumulators

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

Our accumulator pattern is almost always tail recursive

Workshop:
Designing Functions

Adding Numbers by Digits

https://leetcode.com/problems/add-two-numbers/description/