Lists

Concepts of Programming Languages
Lecture 6

CAS CS 320

Announcements

» Assignment 2 is due today!

» Annotated lecture slides have been posted.

» Hope you all signed the course manual.

» Hope you all have installed OCaml.

Outline

Introduce **lists**, look at several examples

Discuss **tail recursion**, in particular its connection to lists

Learn to determine when a function is tail-recursive, and to convert simple recursive implementations to tail recursive implementations

Lists

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

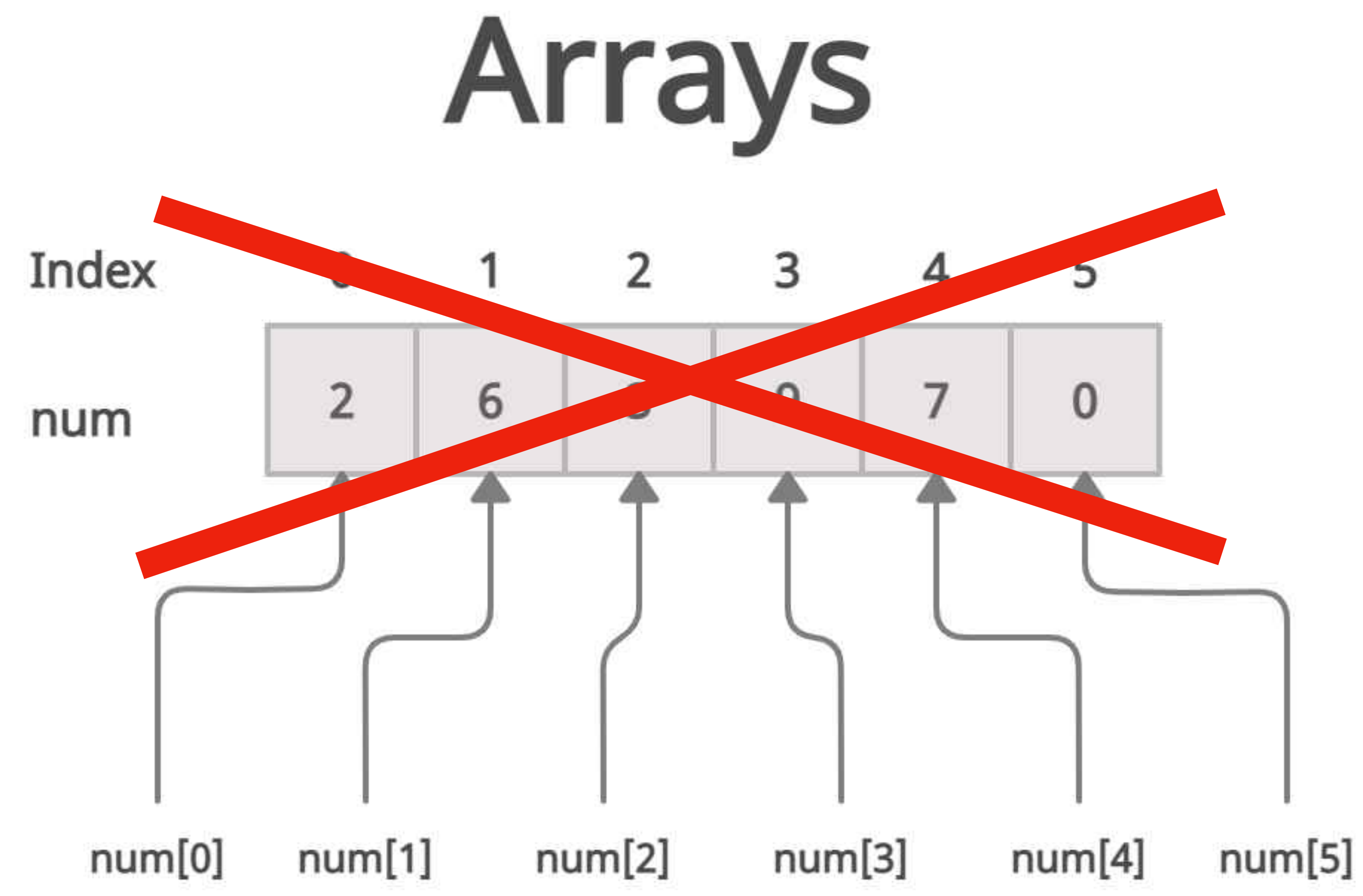
Many important operations on data can be represented as operations on lists (e.g., updating all users in a database)

What is a list not?

A list is *not* an array. We don't have constant-time indexing

A list is *not* mutable. **No data structures in FP are mutable**

(You should think of a list structurally as more like a linked list, sort of)



https://usemynotes.com/what-are-arrays-in-java/

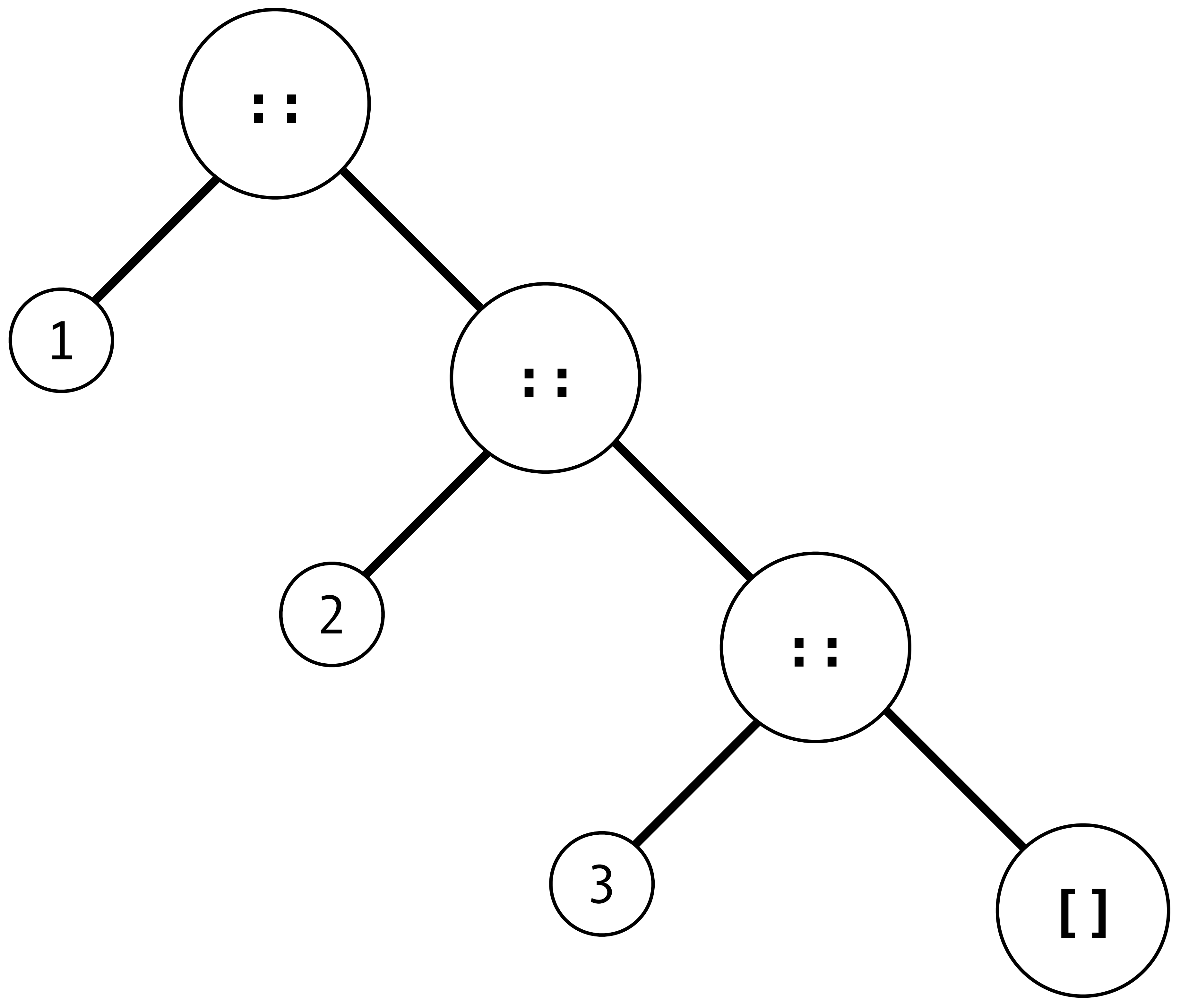
The Picture

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)



Lists (Syntax, Formally)

```
<expr> ::= [ ]
         | <expr> :: <expr>
         | [ <expr> ; <expr> ; ... ; <expr> ]
```

Lists (Syntax, Formally)

```
<expr> ::= []
         | <expr> :: <expr>
         | [ <expr> ; <expr> ; ... ; <expr> ]
```

`[]` is a well-formed expression

Lists (Syntax, Formally)

```
<expr> ::= []
         | <expr> :: <expr>
         | [ <expr> ; <expr> ; ... ; <expr> ]
```

`[]` is a well-formed expression

If e_1 is a well-formed expression and e_2 is a well-formed expression, then e_1 `::` e_2 is a well-formed expression

Lists (Syntax, Formally)

```
<expr> ::= []
         | <expr> :: <expr>
         | [ <expr> ; <expr> ; ... ; <expr> ]
```

[] is a well-formed expression

If e_1 is a well-formed expression and e_2 is a well-formed expression, then e_1 :: e_2 is a well-formed expression

If $e_1, \ldots, e_n$ are well-formed expressions, then [e_1 ; ... ; e_n] is a well-formed expression

List (Syntax, Informally)

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

List (Syntax, Informally)

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

`[]` stands for the empty list (a.k.a. **`nil`**), the list with no elements

List (Syntax, Informally)

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

`[]` stands for the empty list (a.k.a. **`nil`**), the list with no elements

`x :: xs` stands for the list `xs` with `x` prepended to it. The symbol `::` is pronounced "cons" and is a *right associative* operator

List (Syntax, Informally)

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

`[]` stands for the empty list (a.k.a. **`nil`**), the list with no elements

`x :: xs` stands for the list `xs` with `x` prepended to it. The symbol `::` is pronounced "cons" and is a *right associative* operator

`[x1; x2;...; xn]` is a list literal. It's shorthand for a list of a known length

Example

*Construct a function **generate** which, given integers **n**, returns a list consisting of the first **n** positive integers*

And we can make this formal!

Lists (Typing)

$$\frac{}{\Gamma \vdash [\,] : \tau \text{ list}} \text{ (nil)} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \text{ list}}{\Gamma \vdash (e_1 :: e_2) : \tau \text{ list}} \text{ (cons)}$$

The empty list **[]** is of type τ **list** in any context Γ (for any type τ)

If e_1 is of type τ in the context Γ and e_2 is of type τ **list** in the context Γ then $(e_1$ **::** $e_2)$ is of type τ **list** in the context Γ

Homogeneity

$$\frac{}{\Gamma \vdash [\,] : \tau \text{ list}} \text{ (nil)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \text{ list}}{\Gamma \vdash (e_1 :: e_2) : \tau \text{ list}} \text{ (cons)}$$

Notice that this rule enforces that all elements in a list must be the same type

Lists (Semantics, Formally)

$$\frac{}{[] \Downarrow \varnothing}\ \text{(nilEval)} \qquad \frac{e_2 \Downarrow [v_2, \ldots, v_k] \qquad e_1 \Downarrow v_1}{e_1 :: e_2 \Downarrow [v_1, v_2, \ldots, v_k]}\ \text{(consEval)}$$

The empty list **[]** evaluate to the empty list (as a value)

If e_1 evaluates to v_1 and e_2 evaluates to the list value $[v_2, \ldots, v_k]$ then e_1 :: e_2 evaluates to the list value $[v_1, v_2, \ldots, v_k]$

Lists (Semantics, Informally)

$(2 + 3) :: (4 * 12) :: (2 - 1) :: [] \Downarrow [5; 48; 1]$

We evaluate the list $[e_1; e_2; \ldots ; e_k]$ by evaluating each element of the list (from right to left)

Destructing Lists

```
match l with
| [] -> (* something *)
| x :: xs -> (* something else *)
| ... (* other patterns??? *)
```

As with any type in OCaml, we can use pattern matching to destruct lists

With pattern matching, we describe the value we want based on the shape of the list we're matching on

Example 1

*Implement the function **length** where **length l** is the number of elements in **l***

Example 2

Implement the function ***double*** *where* ***double l*** *is the same as the list* ***l*** *but with every element doubled*

Weak Matching on Lists (Syntax)

```
<expr> ::= match <expr> with
         | [] -> <expr>
         | <var> :: <var> -> <expr>
```

If e, e_1, e_2 are well-formed expressions and x, y are valid variable names, then

match e **with | [] ->** e_1 **|** x **::** y **->** e_2

is a well-formed expression

This is "weak" matching because we're not using patterns, we're assuming two fixed branches, e.g. no deep matching

Weak Matching on Lists (Typing)

$$\frac{\Gamma \vdash e : \tau' \ \textsf{list} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma, x : \tau', y : \tau' \ \textsf{list} \vdash e_2 : \tau}{\Gamma \vdash \textsf{match}\ e\ \textsf{with}\ \mid [\,] \to e_1 \mid x :: y \to e_2 : \tau} \ (\textsf{matchList})$$

If e is of type τ' list in the context Γ and e_1 is of type τ in the context Γ and e_2 is of type τ in the context Γ with $(x : \tau')$ and $(y : \tau' \ \text{list})$ added, then the entire match expression is of type τ

Weak Matching on Lists (Typing)

$$\frac{\Gamma \vdash e : \tau' \text{ list} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma, x : \tau', y : \tau' \text{ list} \vdash e_2 : \tau}{\Gamma \vdash \textsf{match } e \textsf{ with } \mid [\,] \to e_1 \mid x :: y \to e_2 : \tau} \ (\text{matchList})$$

Note: Look at how much more compact the rule is!

If e is of type τ' list in the context Γ and e_1 is of type τ in the context Γ and e_2 is of type τ in the context Γ with $(x : \tau')$ and $(y : \tau' \text{ list})$ added, then the entire match expression is of type τ

Weak Matching on Lists (Semantics 1)

$$\frac{e \Downarrow \varnothing \qquad e_1 \Downarrow v}{\textsf{match}\ e\ \textsf{with}\ \mid [\,] \rightarrow e_1 \mid x :: y \rightarrow e_2}\ (\textsf{matchListEvalNil})$$

If e evaluates to the empty list $\varnothing$ and e_1 evaluates to v, then the entire match expression evaluates to v

Weak Matching on Lists (Semantics 2)

$$\frac{e \Downarrow h :: t \qquad e_2' = [t/y][h/x]e_2 \qquad e_2' \Downarrow v}{\textsf{match } e \textsf{ with } \mid [\,] \to e_1 \mid x :: y \to e_2} \text{ (matchListEvalCons)}$$

1. e evaluates to a nonempty list $h :: t$ with first element h and remainder t
2. the expression e_2 with h substituted for x and t substituted for y evaluates to v

implies the entire match statement evaluates to v

Weak Matching on Lists (Semantics 2)

side condition

$$\frac{e \Downarrow h :: t \qquad e_2' = [t/y][h/x]e_2 \qquad e_2' \Downarrow v}{\textsf{match } e \textsf{ with } \mid [\,] \to e_1 \mid x :: y \to e_2} \ (\textsf{matchListEvalCons})$$

1. e evaluates to a nonempty list $h :: t$ with first element h and remainder t

2. the expression e_2 with h substituted for x and t substituted for y evaluates to v

implies the entire match statement evaluates to v

Deep Pattern Matching

```
match <expr> with
| [] -> <expr>
| [h1; h2] -> <expr>
| h1::h2::t -> <expr>
| h::t -> <expr>
| ......
```

Pattern matching is very general. We can match on more complex patterns than just empty and nonempty

Example

Implement the function

delete_every_other : int list -> int list

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. We can create lists of any type by parameterizing the list type (but the elements in one list must all the the same 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. We can create lists of any type by parameterizing the list type (but the elements in one list must all the the same type)

A function is **polymorphic** 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. We can create lists of any type by parameterizing the list type (but the elements in one list must all the the same type)

A function is **polymorphic** if it can be apply to a list parametrized by *any* type

For this, we need *type variables* 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

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. We can't eliminate stack frames until *all* sub-calls finish

Why do we care?

Recursive functions are *expensive* with respect to the call-stack. We can't eliminate stack frames until *all* sub-calls finish

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames, making recursive functions "behave iteratively" when executed

Why do we care?

Recursive functions are *expensive* with respect to the call-stack. We can't eliminate stack frames until *all* sub-calls finish

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames, making recursive functions "behave iteratively" when executed

In Short: Tail-recursive functions are more memory efficient

demo

(summing up numbers in 2 ways)

The Picture

fact 5

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

The Picture

fact 5

fact 4

```
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

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

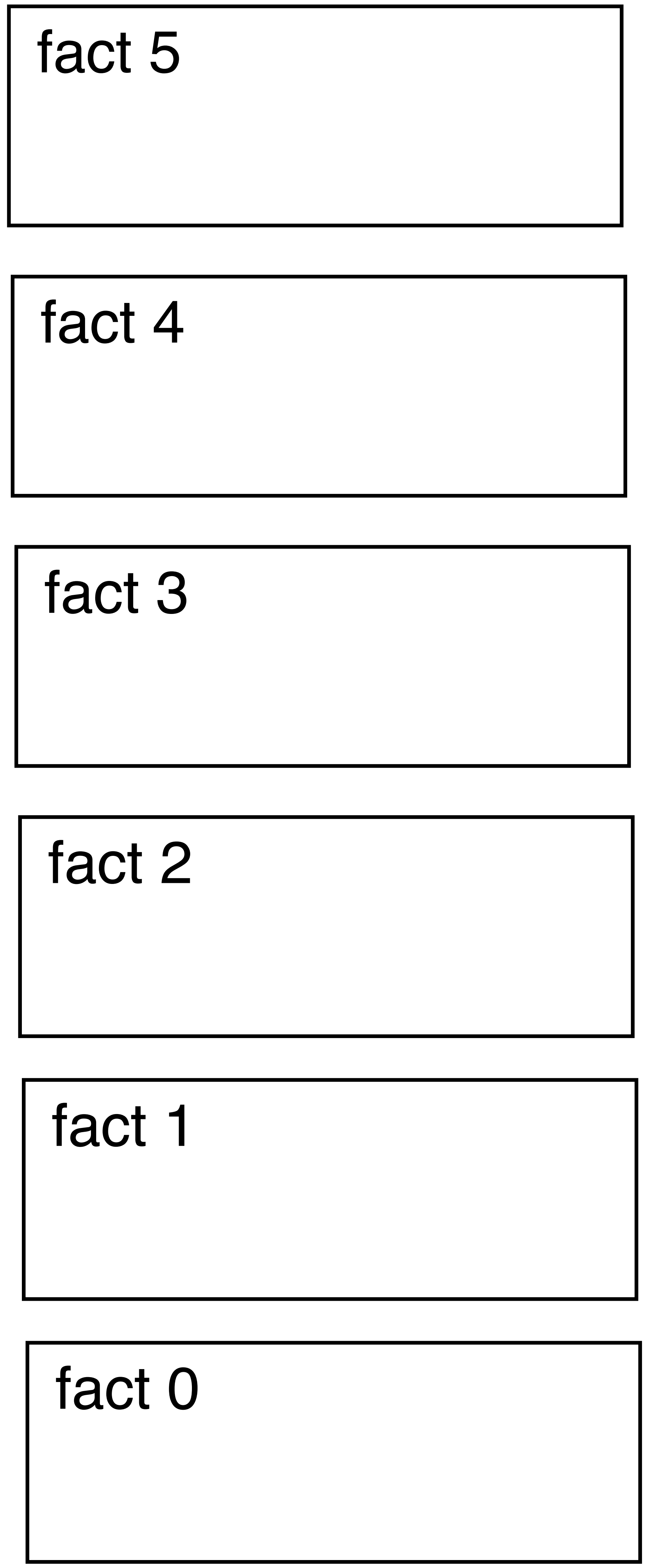
fact 3

fact 2

fact 1

The Picture

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



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

$\Longrightarrow$ **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

$\Longrightarrow$ **3 * 2 = 6**

The Picture

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

fact 5

fact 4

$\Longrightarrow$ **4 * 6 = 24**

The Picture

fact 5

$\Longrightarrow$ **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
$\Longrightarrow$ **5 * 24 = 120**

fact 4
$\Longrightarrow$ **4 * 6 = 24**

fact 3
$\Longrightarrow$ **3 * 2 = 6**

fact 2
$\Longrightarrow$ **2 * 1 = 2**

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

fact 0
$\Longrightarrow$ **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
```

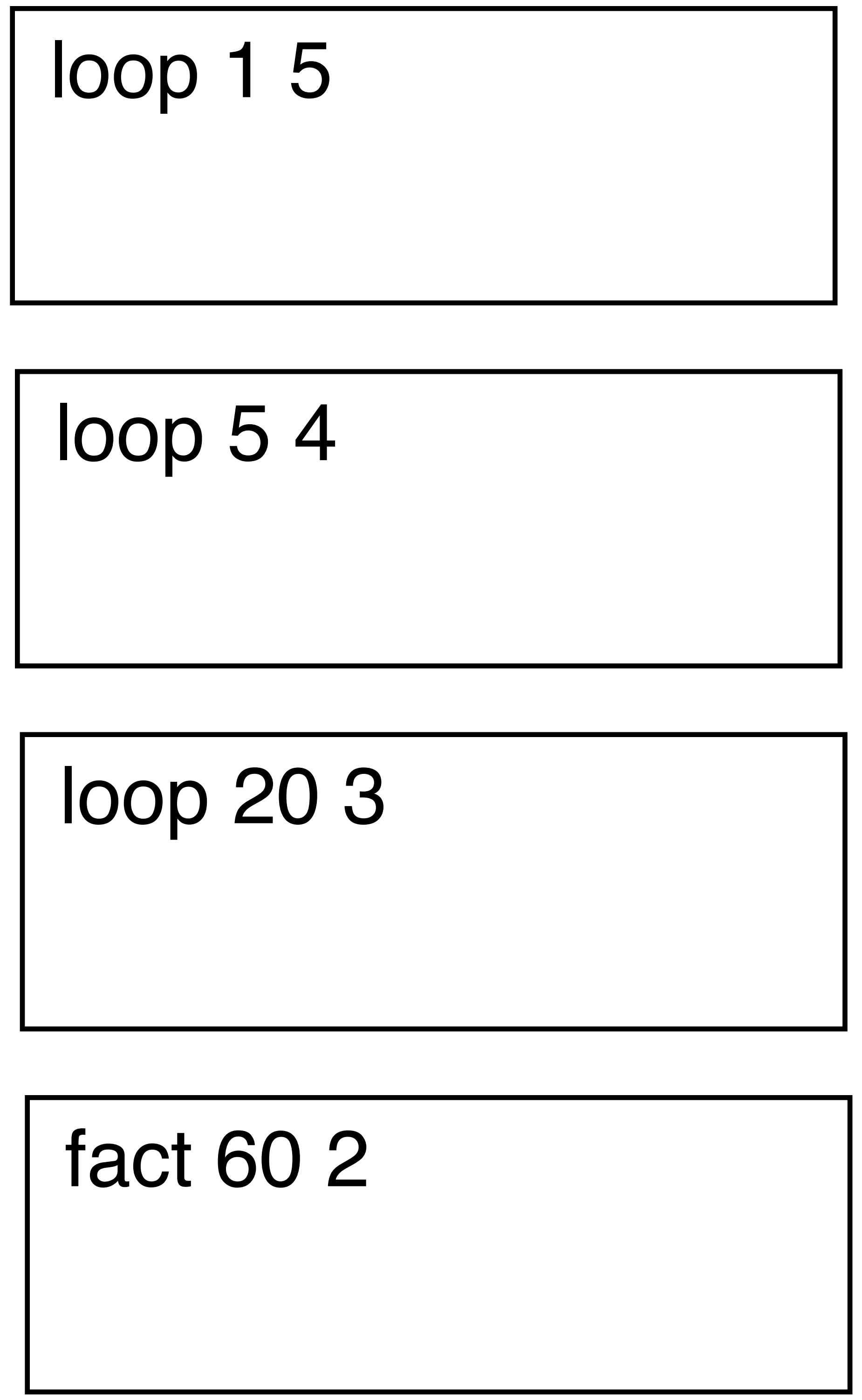
loop 1 5

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
```



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

$\Longrightarrow$ **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

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

$\Longrightarrow$ **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

$\Longrightarrow$ **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

$\Longrightarrow$ **120**

The Picture

loop 1 5

$\Longrightarrow$ **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
```

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:`

* f cannot appear in e1 ... ek

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`

* f cannot appear in e1 ... ek

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**

* f cannot appear in e1 ... ek

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

* f cannot appear in e1 ... ek

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

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

(though it's not the only way to write tail recursive functions)

Example 1 (Tail Recursive)

Implement the function

reverse : 'a list -> 'a list

Example 2 (Tail Recursive)

*Construct a function **generate** that returns a list of the first **n** positive integers*

Example 3 (Tail Recursive)

*Implement the function **length** where **length l** is the number of elements in **l***

Example 4 (Tail Recursive)

Implement the function ***double*** *where* ***double l*** *doubles every element of the list* ***l***

Homework (Tail Recursive)

Implement the function

delete_every_other : ‘a list -> ‘a list

Summary

Lists are used to process collections of homogeneous data

We can use **tail-recursion** to make our implementations more memory efficient, but we have to be careful when working with lists