# CMSC 330:
# Organization of Programming Languages

## Lists and Pattern Matching

Spring 2026

# Let bindings

We use **let** to bind name (identifier) to a value:

```
let x = 100;;   (* x is an immutable binding 100 *)
val x : int = 100
```

Since functions are values, just like ints or strings, let is also used to define functions:

```
let add x y = x + y;;
val add : int -> int -> int
```

# Lists in OCaml

- The basic data structure in Ocaml
  - Lists can be of *arbitrary length*
  - Implemented as a linked data structure
  - Lists must be *homogeneous*
    - All elements have the same type

- Operations
  - Construct lists
  - Destruct them via pattern matching

# Lists Syntax

- `[]` is the empty list (pronounced "nil")
- ***e1*`::`*e2*** prepends element ***e1*** to list ***e2***
  - ***e1*** is the head, ***e2*** is the tail

- `[`***e1***`;`***e2***`;…;`***en***`]` is *syntactic sugar* for ***e1*`::`*e2*`::…::`*en*`::[]`**

## Examples

```
3::[]          (* [3] *)
2::(3::[])     (* [2; 3] *)
[1; 2; 3]      (* 1::(2::(3::[])) *)
```

# Constructing Lists: Evaluation

## Evaluation

- `[]` is a value
- `[`***`e1`***`;…;`***`en`***`]` evaluates to a list of `[`***`v1`***`;…;`***`vn`***`]`
  - **`Where`**
  - ***`e1`*** ⇒ ***`v1`***,
  - …,
  - ***`en`*** ⇒ ***`vn`***

# Constructing Lists: Examples

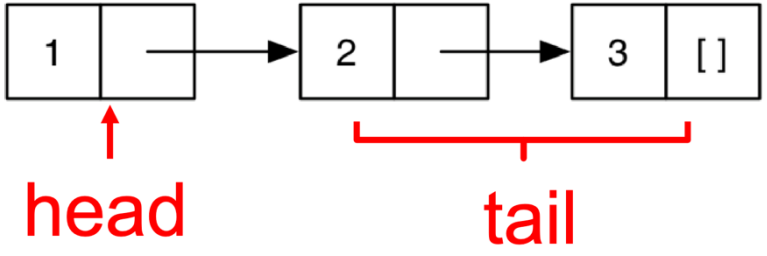
:: operator appends a single item to the front of another list.

```
let y = [1; 1+1; 1+1+1]

let x = 4::y ;;

let z = 5::y ;;

let m = “hello”::”bob”::[];;
```

# List Representation

- A nonempty list is a pair (element, rest of list).
- A list is either
  - The empty list [ ]
  - Or a pair consisting of an element and a list

- **[1;2;3]** is represented as:



# List: Typing

Nil:

`[]: 'a list (* empty list *)`

Cons:

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

If ***e1*** : ***t*** and ***e2*** : ***t*** `list` then ***e1*** `::` ***e2*** : ***t*** `list`

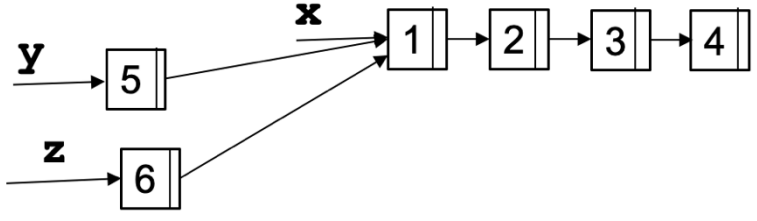
# List Typing Examples

```
let y = 0::[1;2;3]   (* int list *)
let l = [];;         (* 'a list *)
let m = [[1];[2;3]];; (*int list list *)
let w = ["apple"; "banana"; "watermelon"];;

let x = [1;"world"] ;; (* type error *)
```

# Lists are immutable

- Lists are immutable in Ocaml.
- Cannot change an element of a list.
- Instead, create new lists from existing ones

```
let x = [1;2;3;4];;
let y = 5::x;;
let z = 6::x;;
```



# Pattern Matching

- To pull lists apart, use the **`match`** construct
- Syntax

```
match e with
| p1 -> e1
| …
| pn -> en
```

- ***p1***...***pn*** are *patterns*
- ***e1***...***en*** are *branch expressions*

# Pattern Matching Example

```
let is_empty l =
  match l with
    [] -> true
  | (h::t) -> false
```

- Example runs
  - `is_empty []    (* true  *)`
  - `is_empty [1]   (* false *)`
  - `is_empty [1;2] (* false *)`

# Pattern Matching Example (cont.)

```
let hd l =
  match l with
    (h::t) -> h
```

- Example runs
  - `hd [1;2;3]` `(* 1 *)`
  - `hd [2;3]` `(* 2 *)`
  - `hd [3]` `(* 3 *)`
  - `hd []` `(* Exception: Match_failure *)`

# Pattern Matching Example (cont.)

```
let neg n =
  match n with
  |true-> false
  |_-> true
```

```
let is_empty l =
  match l with
[] -> true
   |_-> false
```

- An underscore _ is a wildcard pattern. It matches anything

# "Deep" pattern matching

- `a::b` matches lists with **at least one** element
- `a::[]` matches lists with **exactly one** element
- `a::b::[]` matches lists with **exactly two** elements
- `a::b::c::d` matches lists with **at least three** elements

# Pattern Matching – An Abbreviation

- If there is only one pattern, then

```
let f x = match x with p -> e
```

is shorthand for

```
let f p = e
```

- Example:

```
let fst pair =
    match pair with
    | (x, _) -> x;;
```

```
let fst (x, _) = x;;
```

# Pattern matching is ***AWESOME***

1. You can't forget a case
   - Compiler issues inexhaustive pattern-match warning
2. You can't duplicate a case
   - Compiler issues unused match case warning
3. You can't get an exception
   - Can't do something like **List.hd []**
4. Pattern matching leads to elegant, concise, beautiful code

# Lists and Recursion

- Lists have a recursive structure
  - most functions over lists will be recursive

```
let rec length l = match l with
    [] -> 0
  | (_::t) -> 1 + (length t)
```

- Type of length:
  - `'a list -> int`

# More Examples

- `sum l (* sum of elts in l *)`

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

# More Examples

- `negate l` (* **negate elements in list** *)

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

# More Examples

- `last l  (* last element of l *)`

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

# More Examples

- **append two lists l m**

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

# More Examples

- `rev l  (* reverse list*)`

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

- **rev** takes $O(n^2)$ time.  Can you do better?