Lists

Concepts of Programming Languages
Lecture 4

CAS CS 320

Practice Problem

type shape =
| Rect of { base : float ; height : float }
| Triangle of { sides : float * float ; angle : float }
| Circle of float

```
let area (s : shape) =
  match s with
  | Rect r -> r.base *. r.height
  | Triangle { sides = (a, b) ; angle } -> Float.sin angle *. a *. b
  | Circle r -> r *. r *. Float.pi
```

field punning

Define the variant ***shape*** *which makes this function type-check.*

Outline

- Introduce **lists**, look at several examples
- Discuss **tail recursion**, in particular its connection to lists
- Look at the notion of a **derivation**

Learning Objectives

- » Implement basic functions on lists
- » Determine when a function is tail-recursive, and convert simple recursive implementations to tail recursive implementations
- » Build short typing derivation

Recap

Recall: Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Recall: Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are ordered unlabeled fixed-length heterogeneous collections of data

Recall: Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are ordered unlabeled fixed-length heterogeneous collections of data

(I expect that these are familiar)

Recall: Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are ordered unlabeled fixed-length heterogeneous collections of data

(I expect that these are familiar)

These are useful for returning multiple arguments from a function

Recall: Records

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }

type user = {
  name : string ;
  email : string ;
  num_posts : int ;
}
```

Recall: Records

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }

type user = {
  name : string ;
  email : string ;
  num_posts : int ;
}
```

Records are unordered labeled fixed-length heterogeneous collections of data

Recall: Records

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }

type user = {
  name : string ;
  email : string ;
  num_posts : int ;
}
```

Records are unordered labeled fixed-length heterogeneous collections of data

They are useful for organizing large collections of data (akin to database records)

Recall: Simple Variants

```
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

Type names are **lower_case** and Constructors names are **Upper_case**

Recall: Simple Variants

```
            constructor
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

Type names are **lower_case** and Constructors names are **Upper_case**

Recall: Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Variants can carry data, which allows us to represent more complex structures

Recall: Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Note the syntax

Variants can carry data, which allows us to represent more complex structures

What about variable length data?

Lists

What is a list?

prepend

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

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

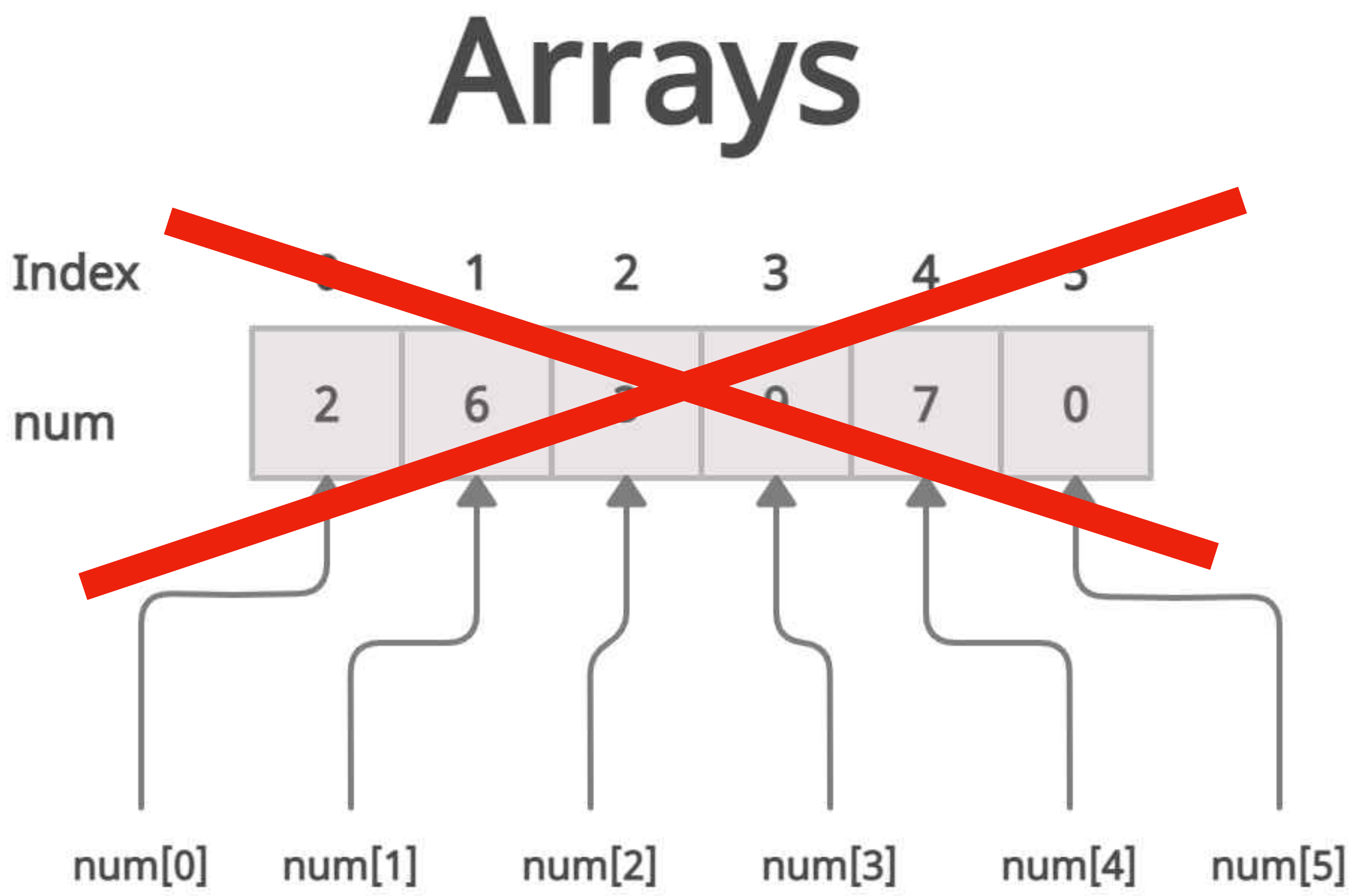
Many important operations on data can be represented as operations on lists (e.g., updating all users in a database)

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/

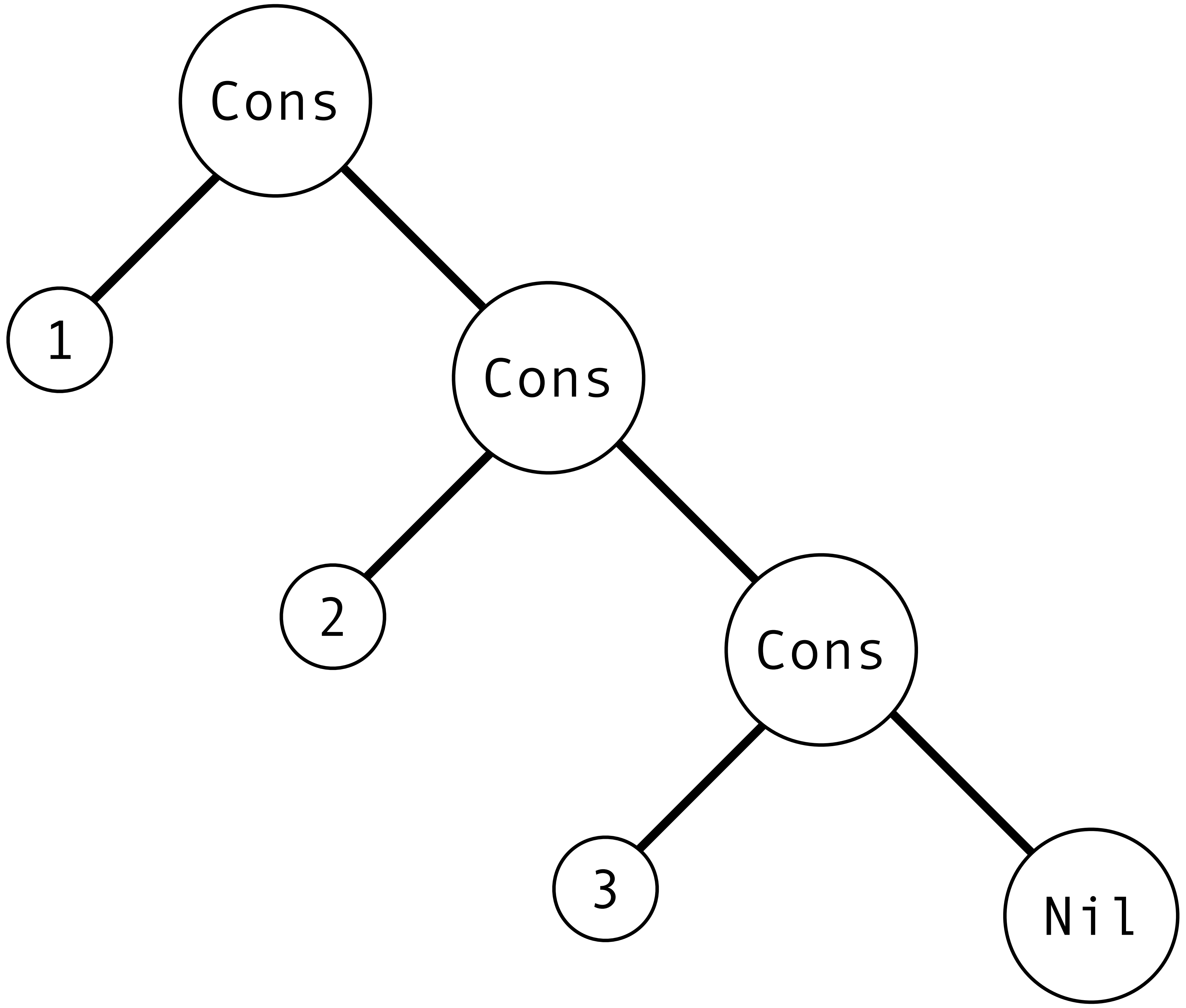
The Picture

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*

Lists (Typing)

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

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

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

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

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

Lists (Semantics, Formally)

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

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`; ... ;` 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

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

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

Reminder: Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

Reminder: Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

```
All data structures in FP are immmutable
```

Reminder: Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

All data structures in FP are imm~~m~~utable

If we want to "update" a list, we have to produce an *entirely new* list

Reminder: Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

All data structures in FP are immmutable

If we want to "update" a list, we have to produce an *entirely new* list

In reality the data is not literally duplicated, there are optimizations which allow for shared data

Practice Problem

Implement the function

remove_all_negatives : int list -> int list

where ***remove_all_negative l*** *is the same as the list* ***l*** *but with all negative numbers removed*

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)

$$\frac{\Gamma \vdash e : \tau' \texttt{ list} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma, x : \tau', y : \tau' \texttt{ list} \vdash e_2 : \tau}{\Gamma \vdash \texttt{match } e \texttt{ with | [] -> } e_1 \texttt{ | } x \texttt{ :: } y \texttt{ -> } e_2 : \tau} \text{(matchList)}$$

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 $(h : \tau')$ and $(t : \tau'$ **list**$)$ added, then the entire match expression is of type τ

Weak Matching on Lists (Semantics 1)

empty list value

$$\frac{e \Downarrow \varnothing \qquad e_1 \Downarrow v}{\texttt{match } e \texttt{ with | [] -> } e_1 \texttt{ | } x \texttt{ :: } y \texttt{ -> } e_2 \Downarrow v} \text{ (matchListEvalNil)}$$

$$\frac{[\,] \Downarrow \varnothing \qquad 2 \Downarrow 2}{(\text{match } [\,] \text{ with } \mid [\,] \to 2 \mid x :: xs \to 0) \Downarrow 2}$$

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)

$$\frac{e \Downarrow h :: t \qquad e_2' = [t/y][h/x]e_2 \qquad e_2' \Downarrow v}{\texttt{match } e \texttt{ with | [] -> } e_1 \texttt{ | } x \texttt{ :: } y \texttt{ -> } e_2 \Downarrow v} \text{(matchListEvalCons)}$$

If

- e_1 evaluates to a nonempty list $h :: t$ with first element h and remainder t
- the expression e_2 with h substituted for x and t substituted for y evaluates to v

then the entire match statement evaluates to v

Weak Matching on Lists (Semantics 2)

side condition

$$\frac{e \Downarrow h :: t \qquad e_2' = [t/y][h/x]e_2 \qquad e_2' \Downarrow v}{\texttt{match}\ e\ \texttt{with}\ \texttt{|}\ \texttt{[]}\ \texttt{->}\ e_1\ \texttt{|}\ x\ \texttt{::}\ y\ \texttt{->}\ e_2 \Downarrow v}\ (\text{matchListEvalCons})$$

If

» e_1 evaluates to a nonempty list $h :: t$ with first element h and remainder t

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

then the entire match statement evaluates to v

Deep Pattern Matching

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

exactly

at least 2 elems.

match top to bottom

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

such that ***delete_every_other l*** *is the first, third, fifth,..., and so on elements of* ***l***

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 uses lists of containing various types (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 uses lists of containing various types (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]` — **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. We can uses lists of containing various types (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

```ocaml
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

Practice Problem

Implement the function

reverse : 'a list -> 'a list

such that ***reverse l*** *is the same as* ***l*** *but in reverse order*

Tail Recursion

demo

(mod 2 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
```

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

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

The Rough Picture

Tail Recursion and Lists

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

Tail Recursion and Lists

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

Be careful with tail-recursive functions on lists.

Tail Recursion and Lists

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

Be careful with tail-recursive functions on lists.

Does the above program concatenate two lists?

Tail Recursion and Lists

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

Be careful with tail-recursive functions on lists.

Does the above program concatenate two lists?

The Moral: Tail recursive functions on lists often reverse the lists

Tail Recursion and Lists

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

should be (List.rev l)

Be careful with tail-recursive functions on lists.

Does the above program concatenate two lists?

The Moral: Tail recursive functions on lists often reverse the lists

Example

Implement the function

reverse : 'a list -> 'a list

in a tail-recursive fashion

Typing Derivations

Recall: Judgements are Statements

$$\{\texttt{b} : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \texttt{string}$$

Recall: Judgements are Statements

$$\{b : \mathtt{bool}\} \vdash \mathtt{if\ b\ then\ 2\ else\ 3} : \mathtt{string}$$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" or "pigs fly" is a statement

Recall: Judgements are Statements

$$\{b : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \boxed{: \texttt{string}}$$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" or "pigs fly" is a statement

We haven't **proved** anything by writing down a typing judgment

Derivations

$$\dfrac{}{\{\} \vdash \mathtt{2} : \mathtt{int}}\ (\text{intLit}) \qquad \dfrac{\dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\ (\text{var}) \qquad \dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\ (\text{var})}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}\ (\text{intAdd})$$

$$\dfrac{\{\} \vdash \mathtt{2} : \mathtt{int} \qquad \{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}{\{\} \vdash \mathtt{let\ y\ =\ 2\ in\ y} + \mathtt{y} : \mathtt{int}}\ (\text{let})$$

Derivations

$$\dfrac{\dfrac{}{\{\}\vdash \mathtt{2} : \mathtt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\mathtt{y} : \mathtt{int}\}\vdash \mathtt{y} : \mathtt{int}}\,(\text{var}) \qquad \dfrac{}{\{\mathtt{y} : \mathtt{int}\}\vdash \mathtt{y} : \mathtt{int}}\,(\text{var})}{\{\mathtt{y} : \mathtt{int}\}\vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}\,(\text{intAdd})}{\{\}\vdash \mathtt{let\ y\ =\ 2\ in\ y} + \mathtt{y} : \mathtt{int}}\,(\text{let})$$

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

Derivations

$$
\dfrac{
  \dfrac{}{\{\} \vdash \texttt{2} : \texttt{int}}\,(\text{intLit})
  \qquad
  \dfrac{
    \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var})
    \qquad
    \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var})
  }{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} + \texttt{y} : \texttt{int}}\,(\text{intAdd})
}{\{\} \vdash \texttt{let y = 2 in } \texttt{y} + \texttt{y} : \texttt{int}}\,(\text{let})
$$

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

Formally, a **derivation** is a tree in which:

Derivations

$$\dfrac{\dfrac{}{\{\} \vdash \texttt{2} : \texttt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var})}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} + \texttt{y} : \texttt{int}}\,(\text{intAdd})}{\{\} \vdash \texttt{let y = 2 in y} + \texttt{y} : \texttt{int}}\,(\text{let})$$

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

Formally, a **derivation** is a tree in which:

» each node is labeled with a typing judgment

Derivations

$$\dfrac{\dfrac{}{\{\} \vdash \mathtt{2} : \mathtt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\text{var}) \qquad \dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\text{var})}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}\,(\text{intAdd})}{\{\} \vdash \mathtt{let\ y\ =\ 2\ in\ y} + \mathtt{y} : \mathtt{int}}\,(\text{let})$$

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

Formally, a **derivation** is a tree in which:

» each node is labeled with a typing judgment

» and typing judgment *follows* from the typing judgments at it's children by an inference rule

Applying Rules

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

$$\frac{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x + 1} : \texttt{int} \qquad \{\texttt{x} : \texttt{int}\} \vdash \texttt{[]} : \texttt{int list}}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{(x + 1) :: []} : \texttt{int list}}\ (\text{cons})$$

So far, we've used rules as ways of describing the behavior of a PL

When we build typing derivations, we *instantiate* the meta-variables in the rule at *particular* expressions, contexts, etc.

Building from the Ground Up

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x} : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{1} : \texttt{int}}\,(\text{intLit})}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x + 1} : \texttt{int}}\,(\text{intAdd}) \qquad \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{[]} : \texttt{int list}}\,(\text{nil})}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{(x + 1) :: []} : \texttt{int list}}\,(\text{cons})$$

Building from the Ground Up

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x} : \texttt{int}}\;(\text{var}) \qquad \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{1} : \texttt{int}}\;(\text{intLit})}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x + 1} : \texttt{int}}\;(\text{intAdd}) \qquad \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{[]} : \texttt{int list}}\;(\text{nil})}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{(x + 1) :: []} : \texttt{int list}}\;(\text{cons})$$

But we can't *just* apply rules, because it's possible that the premises of a rule **also need to be demonstrated**

Building from the Ground Up

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x} : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{1} : \texttt{int}}\,(\text{intLit})}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x + 1} : \texttt{int}}\,(\text{intAdd}) \qquad \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{[]} : \texttt{int list}}\,(\text{nil})}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{(x + 1) :: []} : \texttt{int list}}\,(\text{cons})$$

But we can't *just* apply rules, because it's possible that the premises of a rule **also need to be demonstrated**

This is how we get our tree structure: we apply rules from the ground up

Axioms (When are we done?)

$$
\dfrac{
  \dfrac{
    \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x} : \texttt{int}}\,(\text{var})
    \qquad
    \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{1} : \texttt{int}}\,(\text{intLit})
  }{\{\texttt{x} : \texttt{int}\} \vdash \texttt{x + 1} : \texttt{int}}\,(\text{intAdd})
  \qquad
  \dfrac{}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{[]} : \texttt{int list}}\,(\text{nil})
}{\{\texttt{x} : \texttt{int}\} \vdash \texttt{(x + 1) :: []} : \texttt{int list}}\,(\text{cons})
$$

We know that we can stop building a derivation once we need to derive a premise with an **axiom**, i.e., a rule with no premises

In our case, this will almost always be "literal" or "variable" rules

Integer Literals

(1)

$$\frac{\texttt{n is an int lit}}{\Gamma \vdash \texttt{n} : \texttt{int}}\ (\text{intLit})$$

(2)

$$\frac{\texttt{n is an int lit}}{\texttt{n} \Downarrow n}\ (\text{intLitEval})$$

1. If n is an integer literal, then it is of type int in any context

2. If n is an integer literal, then it evaluates to the number it represents

Float Literals

(1)

$$\frac{\texttt{n is an float lit}}{\Gamma \vdash \texttt{n} : \texttt{float}} \text{ (floatLit)}$$

(2)

$$\frac{\texttt{n is an float lit}}{\texttt{n} \Downarrow n} \text{ (floatLitEval)}$$

1. If n is an float literal, then it is of type float in any context

2. If n is an float literal, then it evaluates to the number it represents

String Literals

(1)

$$\frac{\texttt{s is an string lit}}{\Gamma \vdash \texttt{s} : \texttt{string}}\ (\text{stringLit})$$

(2)

$$\frac{\texttt{s is an string lit}}{\texttt{s} \Downarrow s}\ (\text{stringLitEval})$$

1. If s is an string literal, then it is of type float in any context

2. If s is an string literal, then it evaluates to the string it represents

Boolean Literals

(1)

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\text{trueLit})$$

(2)

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\text{falseLit})$$

(3)

$$\frac{}{\texttt{true} \Downarrow \top}\ (\text{trueLitEval})$$

(4)

$$\frac{}{\texttt{false} \Downarrow \bot}\ (\text{falseLitEval})$$

1. `true` is of type `bool` in any context
2. `false` if of type `bool` in any context
3. `true` evaluates to the value $\top$
4. `false` evaluates to the value $\bot$

Variables

$$\frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau} \text{(intLit)}$$

If v is declared to be of type τ in the context Γ, then v is of type τ in Γ

Variables cannot be evaluated (more on this when we talk about substitution and well-scopedness)

A Note about Side Conditions

$$\dfrac{\dfrac{\dfrac{}{\{\mathtt{x} : \mathtt{int}\} \vdash \mathtt{x} : \mathtt{int}}\ (\text{var}) \qquad \dfrac{}{\{\mathtt{x} : \mathtt{int}\} \vdash \mathtt{1} : \mathtt{int}}\ (\text{intLit})}{\{\mathtt{x} : \mathtt{int}\} \vdash \mathtt{x + 1} : \mathtt{int}}\ (\text{intAdd}) \qquad \dfrac{}{\{\mathtt{x} : \mathtt{int}\} \vdash \mathtt{[]} : \mathtt{int\ list}}\ (\text{nil})}{\{\mathtt{x} : \mathtt{int}\} \vdash \mathtt{(x + 1) :: []} : \mathtt{int\ list}}\ (\text{cons})$$

If a premise is a side-condition this *it is not included in the derivation*

Side conditions need to hold in order to apply the rule, but they don't appear in the derivation itself

Back to the Example

$$\dfrac{\dfrac{}{\{\} \vdash 2 : \texttt{int}}\ (\text{intLit}) \qquad \dfrac{\dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\ (\text{var}) \qquad \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\ (\text{var})}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} + \texttt{y} : \texttt{int}}\ (\text{intAdd})}{\{\} \vdash \texttt{let y = 2 in y} + \texttt{y} : \texttt{int}}\ (\text{let})$$

We need $\{\} \vdash 2 : \texttt{int}$ in order to proof that the bottom typing judgment holds

Now we know that this follows from the **intLit** rule, which says that 2 is always an int, *by fiat*

Semantic Derivations

$$\dfrac{\dfrac{}{\texttt{true} \Downarrow \top}\ (\text{trueEval}) \qquad \dfrac{}{\texttt{2} \Downarrow 2}\ (\text{intEval})}{\texttt{if true then 2 else 3} \Downarrow 2}\ (\text{ifEval})$$

We can also write derivations to prove semantic judgments

The principle is the same, except that the judgments are semantic judgments instead of typing judgments

Example

$\{\} \vdash$ `if true then 2 else 2`$:$`int`

We'll discuss this a lot more

And we'll be giving you a collection of typing rules with examples

Summary

Lists are used to process collections of homogeneous data

We can use **tail-recursion** to make our implementations more memory efficient

We can use **derivations** to *prove* that typing/semantic judgments hold with respect to a collection of inference rules