CS 320: Concepts of Programming Languages

Lecture 5: Lists, Lists, and Lists

Ankush Das

Nathan Mull

Sep 17, 2024

Today's Abstraction

▸ The most important data structure that OCaml provides is a *LIST*

▸ Today, we will learn all there's to know about lists

▸ As usual, we will study syntax, type system, and semantics of lists

▸ (you must be getting tired of this by now)

▸ (*but I will keep repeating this till my last lecture*)

▸ Note: for assignments, please create all files with dummy functions before running "`dune test`"

▸ **Very much like the linked list data structure and not vector**

▸ *Formal Syntax:*

  ▸ `[]` **for an empty list; also called "nil"**

  ▸ `x::l` **for "cons"-ing element `x` to the front of list `l`**

  ▸ `[x_1; x_2; x_3; ...; x_n]` **can also be used to define a fixed list**

Some List Examples

```
'a list
let l1 = []


int list
let l2 = 1::l1


int list
let l3 = 2::3::l2


int list
let l4 = [1; 2; 3]


(* Are l3 and l4 equal? *)
```

```
'a list
let l1 = []

int list
let l2 = 1::l1

int list
let l3 = 2::3::l2

int list
let l4 = [1; 2; 3]

(* Are l3 and l4 equal? *)
```

> **:: is right associative**
>
> **So, this would be equivalent to**
>
> 2::(3::l2)

Example: Generating a List

▸ **Generate a list of first n natural numbers**

```
int -> int list
let rec generate n =
  if n = 0
    then []
    else n::(generate (n-1))

(* generate 5 = [5; 4; 3; 2; 1] *)
```

```
int -> int list
let generate n =
  let rec gen_helper n k =
    if n = 0
      then []
    else k::(gen_helper (n-1) (k+1))
  in
  gen_helper n 1

(* generate 5 = [1; 2; 3; 4; 5] *)
```

- **`[]` can have any type**

$$\overline{\Gamma \vdash [\,] : \alpha \text{ list}}$$

- **If `x` has type τ, then ℓ must have type τ `list` and `x::`ℓ has type τ `list`**

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash \ell : \tau \text{ list}}{\Gamma \vdash (x :: \ell) : \tau \text{ list}}$$

- **All elements of the list must have type τ, then `[e`$_1$`; e`$_2$`; …; e`$_n$`]` has type τ `list`**

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \ldots \qquad \Gamma \vdash e_n : \tau}{\Gamma \vdash [e_1; e_2; \ldots; e_n] : \tau \text{ list}}$$

$$\overline{[\,] \Downarrow [\,]}$$

$$\frac{e \Downarrow v \qquad \ell \Downarrow [v_1;\ v_2;\ \dots v_n]}{e :: \ell \Downarrow [v;\ v_1;\ v_2;\ \dots v_n]}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \dots \qquad e_n \Downarrow v_n}{[e_1;\ e_2;\ \dots e_n] \Downarrow [v_1;\ v_2;\ \dots v_n]}$$

▸ **`[]` is a value**

▸ **If `e` evaluates to `v`, and ℓ evaluates to `[v₁; v₂; …; vₙ]`, then `x::`ℓ evaluates to `[v; v₁; v₂; …; vₙ]`**

▸ **If each element `eᵢ` evaluates to `vᵢ`, then the list evaluates to `[v₁; v₂; …; vₙ]`**

$$[2 + 5; 3 * 3; 4 - 10]$$

$$[2 + 5; 3 * 3; 4 - 10]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1;\ e_2;\ \ldots e_n] \Downarrow [v_1;\ v_2;\ \ldots v_n]}$$

$$[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; \ e_2; \ \ldots e_n] \Downarrow [v_1; \ v_2; \ \ldots v_n]}$$

$$[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; \ e_2; \ \ldots e_n] \Downarrow [v_1; \ v_2; \ \ldots v_n]}$$

$$2 + 5 \Downarrow 7$$

$$[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; \ e_2; \ \ldots e_n] \Downarrow [v_1; \ v_2; \ \ldots v_n]}$$

$$2 + 5 \Downarrow 7 \qquad\qquad 3 * 3 \Downarrow 9$$

$$\overline{[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1;\ e_2;\ \ldots e_n] \Downarrow [v_1;\ v_2;\ \ldots v_n]}$$

$$2 + 5 \Downarrow 7 \qquad\qquad 3 * 3 \Downarrow 9 \qquad\qquad 4 - 10 \Downarrow -6$$

$$\overline{[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; e_2; \ldots e_n] \Downarrow [v_1; v_2; \ldots v_n]}$$

$$2 + 5 \Downarrow 7 \qquad 3 * 3 \Downarrow 9 \qquad 4 - 10 \Downarrow -6$$

$$[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; \ e_2; \ \ldots e_n] \Downarrow [v_1; \ v_2; \ \ldots v_n]}$$

$$\frac{2 \Downarrow 2 \qquad 5 \Downarrow 5 \qquad 7 = 2 + 5}{2 + 5 \Downarrow 7} \qquad 3 * 3 \Downarrow 9 \qquad 4 - 10 \Downarrow -6$$

$$\frac{}{[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; e_2; \ldots e_n] \Downarrow [v_1; v_2; \ldots v_n]}$$

$$\frac{2 \Downarrow 2 \qquad 5 \Downarrow 5 \qquad 7 = 2 + 5}{2 + 5 \Downarrow 7}$$

$$3 * 3 \Downarrow 9$$

$$4 - 10 \Downarrow -6$$

$$\frac{}{[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 * v_2}{e_1 * e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; \ e_2; \ \ldots e_n] \Downarrow [v_1; \ v_2; \ \ldots v_n]}$$

$$\frac{2 \Downarrow 2 \qquad 5 \Downarrow 5 \qquad {\color{red}7 = 2 + 5}}{2 + 5 \Downarrow 7} \qquad \frac{3 \Downarrow 3 \qquad 3 \Downarrow 3 \qquad {\color{red}9 = 3 * 3}}{3 * 3 \Downarrow 9} \qquad 4 - 10 \Downarrow -6$$

$$[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 * v_2}{e_1 * e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \dots \qquad e_n \Downarrow v_n}{[e_1; e_2; \dots e_n] \Downarrow [v_1; v_2; \dots v_n]}$$

$$\frac{2 \Downarrow 2 \qquad 5 \Downarrow 5 \qquad 7 = 2 + 5}{2 + 5 \Downarrow 7} \qquad \frac{3 \Downarrow 3 \qquad 3 \Downarrow 3 \qquad 9 = 3 * 3}{3 * 3 \Downarrow 9} \qquad 4 - 10 \Downarrow -6$$

$$[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 * v_2}{e_1 * e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \ldots \qquad e_n \Downarrow v_n}{[e_1; e_2; \ldots e_n] \Downarrow [v_1; v_2; \ldots v_n]} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 - v_2}{e_1 - e_2 \Downarrow v}$$

$$\frac{2 \Downarrow 2 \qquad 5 \Downarrow 5 \qquad 7 = 2 + 5}{2 + 5 \Downarrow 7} \qquad \frac{3 \Downarrow 3 \qquad 3 \Downarrow 3 \qquad 9 = 3 * 3}{3 * 3 \Downarrow 9} \qquad \frac{4 \Downarrow 4 \qquad 10 \Downarrow 10 \qquad -6 = 4 - 10}{4 - 10 \Downarrow -6}$$

$$[2 + 5; 3 * 3; 4 - 10] \Downarrow [7; 9; -6]$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 * v_2}{e_1 * e_2 \Downarrow v} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \dots \qquad e_n \Downarrow v_n}{[e_1; e_2; \dots e_n] \Downarrow [v_1; v_2; \dots v_n]} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 - v_2}{e_1 - e_2 \Downarrow v}$$

▸ **So far, we have seen how to construct lists**

▸ **But, how do we use lists? (there's no point building them if we can't use them)**

▸ **To use lists, we go back to our new best friend: pattern matching**

▸ **Remember, lists have two possibilities: '`nil`' (or `[]`) and '`cons`' (or `h::t`) where `h` is the head of the list and `t` is the tail of the list**

▸
```
match <expr> with
| [] -> <expr>
| h::t -> <expr>
| ……
```

```
'a list -> int
let rec length lst =
  match lst with
  | [] -> 0
  | h::t -> 1 + (length t)
```

‣ **If the list is empty, i.e., `[]`, then return `0`**

‣ **Else, add `1` to the length of `t`**

‣ **Note the type: what is `'a list`?**

▸ **What should be the type of `lst`?**

▸ **Should it be `int list`? Or `bool list`? Or something else?**

▸ **Technically, it can be `α list` for any type `α`**

▸ **In OCaml, this is written as `'a list`**

▸ **These functions are called polymorphic functions (similar to generics in Java, etc.)**

▸ **These functions are defined for all types and can be used at any type.**

▶ **This length function can be applied to any argument**

▶ `length [1; 2; 3] = 3`
  `length [4.; 1.; 2.; 5.] = 4`
  `length ["cs320", "cs599"] = 2`

▶ **In fact, they only need to be defined once and can be called with different types without defining them individually for each type**

▶ **This makes them powerful! They are crucial for data structures like stacks, queues, trees that can store arbitrary data**

```
int list -> int
let rec sum lst =
  match lst with
  | [] -> 0
  | h::t -> h + sum t
```

‣ **If the list is empty, i.e., `[]`, then return `0`**

‣ **Else, add `h` to the sum of `t`**

‣ *Challenge: Write a function that can sum up an arbitrary list type*

▸ **Like every type so far, lists are also immutable**

▸ **We create new lists from the older list**

▸ **For example:**

```
int list -> int list
let rec inc lst =
  match lst with
  | [] -> []
  | h::t -> (h+1)::(inc t)
```

▸ **Calling `inc` does not mutate the argument**

$$\frac{\Gamma \vdash e : \tau \text{ list} \qquad \Gamma \vdash e_1 : \tau' \qquad \Gamma, h : \tau, t : \tau \text{ list} \vdash e_2 : \tau'}{\Gamma \vdash \text{match } e \text{ with } [] \rightarrow e_1 \mid h :: t \rightarrow e_2 : \tau'}$$

▸ **e must have type** τ `list`

▸ **Each branch should have the same type: e₁ has type** τ'

▸ **For the second branch, add** `h : `τ **and** `t : `τ `list` **to the context**

▸ **In this context, e₂ must have type** τ'

$$\frac{e \Downarrow [\,] \qquad e_1 \Downarrow v_1}{(\text{match } e \text{ with } [\,] \to e_1 \mid h :: t \to e_2) \Downarrow v_1}$$

$$\frac{e \Downarrow (v :: vs) \qquad [v/h]\,[vs/t]e_2 \Downarrow v_2}{(\text{match } e \text{ with } [\,] \to e_1 \mid h :: t \to e_2) \Downarrow v_2}$$

▸ **Since `e : τ list`, it can either evaluate to an empty list or a non-empty list (*a consequence of preservation*)**

▸ **First rule handles empty case, Second rule handles non-empty case**

- `match <expr> with`

  `| [] -> <expr>`

  `| [h1; h2] -> <expr>`

  `| h1::h2::t -> <expr>`

  `| h::t -> <expr>`

  `| ……`

▸ `match <expr> with`

`| [] -> <expr>`

`| [h1; h2] -> <expr>`

`| h1::h2::t -> <expr>`

`| h::t -> <expr>`

`| ……`

[] for empty list

▸ `match <expr> with`

    `| [] -> <expr>`

    `| [h1; h2] -> <expr>`

    `| h1::h2::t -> <expr>`

    `| h::t -> <expr>`

    `| ……`

`[]` **for empty list**

`[h1; h2]` **for a list of size exactly 2**

▶ `match <expr> with`

`| [] -> <expr>`

`| [h1; h2] -> <expr>`

`| h1::h2::t -> <expr>`

`| h::t -> <expr>`

`| ......`

`[]` for empty list

`[h1; h2]` for a list of size exactly 2

`h1::h2::t` for a list of size at least 2

▸ `match <expr> with`

`| [] -> <expr>`

> **[] for empty list**

`| [h1; h2] -> <expr>`

> **[h1; h2] for a list of size exactly 2**

`| h1::h2::t -> <expr>`

> **h1::h2::t for a list of size at least 2**

`| h::t -> <expr>`

`| ......`

> **h::t for a list of size at least 1**

▸ **Recall that programmers can define their own types in OCaml**

▸ **List is just one such type; can be defined as follows:**

```
type intlist =
  | Nil
  | Cons of int * intlist
```

▸ **This is a data-carrying variant; a value of list type can either be `Nil` or `Cons` of an integer (head) and another list (tail)**

▸ **The built-in list type just uses `[]` for `Nil` and `::` for `Cons`**

```
type intlist =
  | Nil
  | Cons of int * intlist
```

▸ **Empty list can be created using:**

```
intlist
let x = Nil
```

▸ **Bigger lists can be created using:**

```
intlist
let x = Cons (1, (Cons (2, (Cons (3, Nil)))))
```

▸ ```
match <expr> with
| Nil -> <expr>
| Cons(h, t) -> <expr>
| ……
```

▸ **What should be the typing rule?**

$$\frac{\Gamma \vdash e : \mathsf{int\ list} \qquad \Gamma \vdash e_1 : \tau' \qquad \Gamma, h : \mathsf{int}, t : \mathsf{int\ list} \vdash e_2 : \tau'}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{Nil} \to e_1 \mid \mathsf{Cons}(h, t) \to e_2 : \tau'}$$

▸ *Homework: Write down the semantics rules for this expression*

```
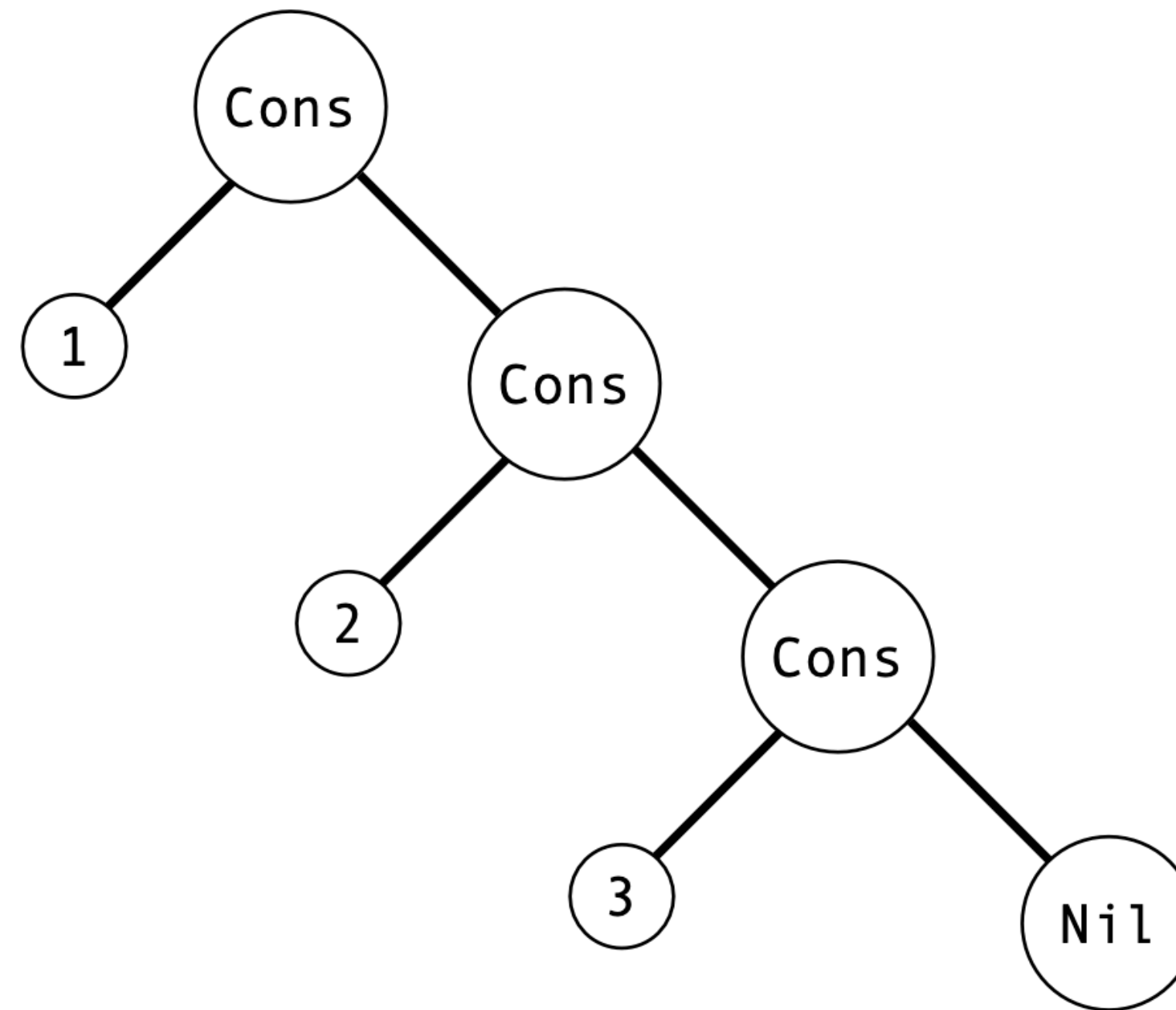match <expr> with
| Nil -> <expr>
| Cons(h, Nil) -> <expr>
| Cons(h1, Cons(h2, Nil)) -> <expr>
| Cons(h1, Cons(h2, t)) -> <expr>
| Cons(h, t) -> <expr>
| ……
```

▸ *Homework: Write down the typing and semantics rules for this expression*

```
intlist
let x = Cons (1, (Cons (2, (Cons (3, Nil)))))
```

```
type intlist = { head : int; tail : intlist}
```

▸ **But I don't like this representation. Tell me why?**

▸ **How do you represent an empty list?**

▸ **How about this instead? Types can also be mutually recursive**

```
type intlist = { head : int; tail : intlist_option }

and intlist_option =
  | None
  | Some of intlist
```

How to Represent All Lists?

▸ The answer is *polymorphism*!

▸ Like functions, types can also be polymorphic

▸ Main idea: element inside a list can be of "*any*" type; "*any*" types are represented using `'a`, `'b`, `'c`, ……

▸ Let's do some examples!

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
int list
let x = Cons (1, Cons (2, Nil))
```

Creating Specific Typed Lists

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

int list
```
let x = Cons (1, Cons (2, Nil))
```

string list
```
let y = Cons ("One", Cons ("Two", Nil))
```

float list
```
let z = Cons (1., Cons (2., Nil))
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
'a list -> int
let rec length lst =
  match lst with
  | Nil -> 0
  | Cons(_, t) -> 1 + length t
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
'a list -> int
let rec length lst =
  match lst with
  | Nil -> 0
  | Cons(_, t) -> 1 + length t
```

```
'a list -> 'a list
let rec swap2 lst =
  match lst with
  | Nil -> Nil
  | Cons(h1, Cons(h2, t)) -> Cons (h2, Cons (h1, swap2 t))
  | Cons(h, Nil) -> Cons(h, Nil)
```

Can Size be Polymorphic?

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
let rec sum lst            =
  match lst with
  | Nil ->
  | Cons(h, t) ->
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
let rec sum lst              =
  match lst with
  | Nil -> zero
  | Cons(h, t) ->
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
let rec sum lst zero      =
  match lst with
  | Nil -> zero
  | Cons(h, t) ->
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
let rec sum lst zero add =
  match lst with
  | Nil -> zero
  | Cons(h, t) ->
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
let rec sum lst zero add =
  match lst with
  | Nil -> zero
  | Cons(h, t) -> add h (sum t zero add)
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
'a list -> 'b -> ('a -> 'b -> 'b) -> 'b
let rec sum lst zero add =
  match lst with
  | Nil -> zero
  | Cons(h, t) -> add h (sum t zero add)
```

```ocaml
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```ocaml
(* 'a list -> 'b -> ('a -> 'b -> 'b) -> 'b *)
let rec sum lst zero add =
  match lst with
  | Nil -> zero
  | Cons(h, t) -> add h (sum t zero add)
```

```ocaml
(* int -> int -> int *)
let add_int a b = a + b

(* int list -> int *)
let sum_int lst = sum lst 0 add_int
```

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
'a list -> 'b -> ('a -> 'b -> 'b) -> 'b
let rec sum lst zero add =
  match lst with
  | Nil -> zero
  | Cons(h, t) -> add h (sum t zero add)
```

```
int -> int -> int
let add_int a b = a + b

int list -> int
let sum_int lst = sum lst 0 add_int
```

```
float -> float -> float
let add_float a b = a +. b

float list -> float
let sum_float lst = sum lst 0. add_float
```

```
type 'a option =
  | None
  | Some of 'a
```

```
'a list -> 'a option
let head lst =
  match lst with
  | Nil -> None
  | Cons(h, _) -> Some h
```

▸ **This may or may not hold a value; useful when not all inputs have a valid output; like the head function cannot return a value when list is empty**

▸ **Some people use *null pointers* for this. Please don't EVER use null pointers**

▸ **Null pointers were called the "*billion-dollar mistake*" by Tony Hoare, the person who invented them**

```
type ('a, 'e) result =
  | Ok of 'a
  | Err of 'e
```

```
'a list -> ('a, string) result
let head lst =
  match lst with
  | Nil -> Err "Empty list has no head"
  | Cons(h, _) -> Ok h
```

▸ **Polymorphic types can have multiple parameters**

▸ **Allows users to return a result or an error message**

‣ **Some recursive functions can be made *tail recursive***

‣ **What is tail recursive?**

‣ **A recursive function is tail recursive when you do not perform any computation on the result of the recursive call**

‣ ```
let rec factorial n =
if n = 0 then 1 else n * factorial (n-1)
```

‣ ***Not Tail Recursive:* The result of the recursive call to factorial is multiplied with "n"**

‣ **Can it be made tail recursive? Yes!**

```
int -> int
let factorial n =
  let rec fact_helper n acc =
    if n = 0 then acc else fact_helper (n-1) (acc*n)
  in
  fact_helper n 1
```

▸ **Make the result another argument**

▸ **`acc` (short for accumulator) stores the result of factorial; when `n` reaches 0, we simply return `acc`**

▸ **In the else branch, the recursive call is directly returned**

```
'a list -> 'a list
let rev lst =
    let rec rev_helper lst acc =
        match lst with
        | Nil -> acc
        | Cons(h, t) -> rev_helper t (Cons(h, acc))
    in
    rev_helper lst Nil
```

▸ **Can every recursive function be made tail recursive? Theoretically speaking, yes! There is an entire PL theory called "*Continuation Passing Style*" based on this. Much harder in practice though**

▸ *Homework: Use reverse to implement list append using tail recursion*

Homework

▸ **Read about records**

▸ **Practice as many typing derivations and semantics derivations as you can! Please!**

▸ **Write out the typing rules for other destructors for tuples**