Lecture 32:

- Quiz 7
- OCaml: Lists, More Functions

Announcements:

- HW-6 due
- HW-7 out soon ...

OCaml Lists

Lists in OCaml take the form: [e_1; e_2; $\cdots$; e_n]

```
# [1; 2; 3; 4] ;;
- : int list = [1; 2; 3; 4]
```

- Lists can be of any length (including empty [])
- All values in a list must be of the **same type** ("homogeneous")

```
# [1; 2; false] ;;
Error: This expression has type bool but an expression was
       expected of type int

# [[1; 2]; [3; 4]] ;;
- : int list list = [[1; 2]; [3; 4]]

# [[1.; 2.]; [3.; 4.; 5.]; []] ;;
- : float list list = [[1.; 2.]; [3.; 4.; 5.]; []]
```

- the last two examples are "nested lists"
- note the types are inferred by OCaml!

OCaml Lists

List types ...

- a list of ints is an "int list", written `int list`
- a list of lists of ints is written `int list list` (i.e., an "(int list) list")

What is the type of the empty list?

```
# [] ;;
- : 'a list = []
```

- a list of *any* type, denoted `'a`
- you can think of `'a` as a **type variable**
- sometimes written as α (e.g., an α-list)
- where greek letters are used to denote type variables

We'll see more examples of type variables soon ...

OCaml Lists

List append (`@`) ... "concatenation"

- Returns an entirely new list, where
- ... values in the second are appended to the values of the first

```
# [1; 3] @ [3; 4] ;;
- : int list = [1; 3; 3; 4]

# ['a'; 'b'] @ [] ;;
- : char list = ['a'; 'b']
```

Append is a generic (polymorphic) function that works over lists of any type:

```
# (@) ;;
- : 'a list -> 'a list -> 'a list = <fun>
```

- takes two lists of the same type α, returns a new α-list
- in other words `(@)` : α-list $\rightarrow$ α-list $\rightarrow$ α-list

OCaml Lists

OCaml supports "**partial function application**"

```
# (@) [1; 2] ;;
- : int list -> int list = <fun>
```

- Note the type is now int-list → int-list ... why?

```
# let prepend_1 = (@) [1] ;;
val prepend_1 : int list -> int list = <fun>

# prepend_1 [2; 3] ;;
- : int list = [1; 2; 3]
```


OCaml Lists

List construction (::) ... aka "*cons*"

- Creates new list from a value and a list ... a list as a sequence of cons

```
# 1 :: [2; 3] ;;
- : int list = [1; 2; 3]

# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]

# List.cons ;;       (* similar to (::) ... *)
- : 'a -> 'a list -> 'a list = <fun>
```

- Thus cons takes an α value and an α-list and returns a new α-list

Q: Is cons (::) right or left associative? ... Right associative!

- `1::(2::(3::[]))`
- `1::2` is a type error since second operand is not a list!