Question 1: How many arguments does this function have?
let rec append = function
| ([], ys)       -> ys
| (x::xs, ys)   -> x :: append (xs,ys)
One (the argument is a tuple)

Question 2: Give an example of a recursive datatype.

binary tree

Question 3: What is the depth of a balanced binary search tree with
n elements?

O (log n)
Functions as Values - Intro

• Powerful technique: treat **functions as data**

• Idea: functions may be arguments or results of other functions.

  Compare to a familiar concept: \( \int \sin(x) \, dx \)

• Examples:
  • comparison to be used in sorting
  • numerical function to be integrated
  • the tail of an infinite list! (to be seen later)

• Also: **higher-order** function or a **functional**: a function that operates on other functions (e.g.: map)
In OCaml, functions can be

- passed as *arguments* to other functions,
- returned as *results*,
- put into lists, tree, etc.:

```ocaml
[(fun n -> n * 2); (fun n -> n * 3); (fun k -> k + 1)];
```

- : (int -> int) list = [<fun>; <fun>; <fun>]

- but **not** tested for equality.
fun x ->  

\[ E \] is the function \( f \) such that \( f(x) = E \)

The function \((\text{fun } n \rightarrow n \times 2)\) is a *doubling function*.

\[(\text{fun } n \rightarrow n \times 2)\];

\[- : \text{int} \rightarrow \text{int} = \langle \text{fun} \rangle\]

\[(\text{fun } n \rightarrow n \times 2) \ 17;\]

\[- : \text{int} = 34\]
Functions without Names

In: (fun n -> n * 2) 2;;
Out: - : int = 4

... can be given a name by a let declaration

In: let double = fun n -> n * 2;;
Out: val double : int -> int = <fun>

In: let double n = n * 2;;
Out: val double : int -> int = <fun>

In both cases:

In: double 2;
Out: - : int = 4
Functions without Names

function can be used for pattern-matching:

\[
\text{function } P_1 \rightarrow E_1 \mid \ldots \mid P_n \rightarrow E_n
\]

for example:

\[
\text{function } 0 \rightarrow \text{true} \mid _{-} \rightarrow \text{false}
\]

which is equivalent to:

\[
\text{fun } x \rightarrow \text{match } x \text{ with } 0 \rightarrow \text{true} \mid _{-} \rightarrow \text{false}
\]

\[
\text{let is_zero = fun } x \rightarrow \text{match } x \text{ with } 0 \rightarrow \text{true} \mid _{-} \rightarrow \text{false}
\]

\[
\text{let is_zero = function } 0 \rightarrow \text{true} \mid _{-} \rightarrow \text{false}
\]
• A function can only have **one** argument

• Two options for **multiple** arguments:
  1. tuples (e.g., pairs) *as seen in previous lectures*
  2. a function that returns another function as a result
     
     → this is called **currying** (after H. B. Curry) \(^1\)

• Currying: expressing a function taking multiple arguments as **nested functions**.

\(^1\) Credited to Schönfinkel, but *Schönfinkeling* didn’t catch on…
Curried Functions

Taking multiple arguments as **nested** functions, so, instead of:

\[
\text{In : } \text{fun} \ (n, \ k) \to n \times 2 + k;;
\]
\[
\text{Out: } - : \text{int} \times \text{int} \to \text{int} = \text{<fun>}
\]

We can **nest** the \text{fun}-notation:

\[
\text{In : } \text{let} \ it = \text{fun} \ k \to (\text{fun} \ n \to n \times 2 + k);;;
\]
\[
\text{Out: } \text{val} \ it : \text{int} \to \text{int} \to \text{int} = \text{<fun>}
\]

\[
\text{In : } it \ 1 \ 3;
\]
\[
\text{Out: } - : \text{int} = 7
\]
A *curried function* returns another function as its result.

```ocaml
let prefix = (fun a -> (fun b -> a ^ b))
val prefix : string -> string -> string = <fun>
```

prefix yields functions of type `string -> string`.

```ocaml
let promote = prefix "Professor ";
val promote : string -> string = <fun>
```

```ocaml
promote "Mopp";;
- : string = "Professor Mopp"
```
Shorthand for Curried Functions

A function-returning function is just a function of two arguments.

A function over pairs has type \((\sigma_1 \times \sigma_2) \rightarrow \tau\).
A curried function has type \(\sigma_1 \rightarrow (\sigma_2 \rightarrow \tau)\).

This curried function is nicer than nested \texttt{fun} binders:

\begin{verbatim}
let prefix a b = a ^ b;;
val prefix : string -> (string -> string)
\end{verbatim}

Syntax: the symbol \(\rightarrow\) associates to the right

\begin{verbatim}
fun x_1 x_2 ... x_n -> E    let f x_1 x_2 ... x_n = E

let dub = prefix "Sir ";;
val dub : string -> string = <fun>
\end{verbatim}

Curried functions allows partial application (to the first argument).
Partial Application: A Curried Insertion Sort

Key question: How to generalize <= to any data type?

```
let rec insort lessequal =
  let rec ins = function
  | x, [] -> [x]
  | x, y::ys ->
    if lessequal x y then x::y::ys
    else y :: ins (x, ys)
  in
  let rec sort = function
  | [] -> []
  | x::xs -> ins (x, sort xs)
  in
  sort
```

val insort : ('a -> 'a -> bool) -> 'a list -> 'a list
Partial Application: A Curried Insertion Sort

Note: \((<=)\) denotes comparison operator as a function

In : \text{insort} (<=) \[5; 3; 9; 8]\;;
Out: \(\text{- : int list = } [3; 5; 8; 9]\)

In : \text{insort} (\geq) \[5; 3; 9; 8]\;;
Out: \(\text{- : int list = } [9; 8; 5; 3]\)

In : \text{insort} (<=) \["bitten"; "on"; "a"; "bee"];;
Out: \(\text{- : string list = } ["a"; "bee"; "bitten"; "on"]\)
**map: the ‘Apply to All’ Functional**

note: built-in as List.map

```
let rec map f = function
| []   -> []
| x::xs -> (f x) :: map f xs
```

In : map (fun s -> s ^ "ppy");
Out: - : string list -> string list = <fun>

In : map (fun s -> s ^ "ppy") ["Hi"; "Ho"];;
Out: - : string list = ["Hippy"; "Hoppy"]

In : map (map double) [[1]; [2; 3]];;
Out: - : int list list = [[2]; [4; 6]]
Example: Matrix Transpose

\[
\begin{pmatrix}
a & b & c \\
d & e & f \\
\end{pmatrix}
\end{pmatrix}^T
= \begin{pmatrix}
a & d \\
b & e \\
c & f \\
\end{pmatrix}
\]

let rec transp = function
| [] :: _ -> []
| rows -> (map List.hd rows) ::
            (transp (map List.tl rows))
Example: Matrix Transpose

let rec transp = function
 | []::_ -> []
 | rows  -> (map List.hd rows) ::
               (transp (map List.tl rows))

In : let rows = [[1; 2; 3]; [4; 5; 6]];;

In : List.hd;;
Out: - : 'a list -> 'a = <fun>
In : transp;
Out: - : 'a list list -> 'a list list

In : map List.hd rows;
Out: - : int list = [1; 4]
In : map tl rows;
Out: - : int list list = [[2; 3]; [5; 6]]
In : transp rows;
Out: - : int list list = [[1; 4]; [2; 5]; [3; 6]]
Review of Matrix Multiplication

\[
\begin{pmatrix}
A_1 & \cdots & A_k
\end{pmatrix}
\cdot
\begin{pmatrix}
B_1 \\
\vdots \\
B_k
\end{pmatrix}
= 
\left( A_1 B_1 + \cdots + A_k B_k \right)
\]

The right side is the vector dot product \( \vec{A} \cdot \vec{B} \)

Repeat for each row of \( A \) and column of \( B \)
Review of Matrix Multiplication

\[ A \cdot B = \begin{pmatrix} 2 & 0 \\ 3 & -1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 2 \\ 4 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 4 \\ -1 & 1 & 6 \\ 4 & -1 & 0 \\ 5 & -1 & 2 \end{pmatrix} \]

For element \((i,j)\) of \(A \times B\):

- dot-product of row \(i\) and column \(j\)
Matrix Multiplication in OCaml

Dot product of two vectors—a curried function

let rec dotprod xs ys =  
  match xs, ys with  
  | [], [] -> 0.0  
  | x::xs, y::ys -> x *. y + dotprod xs ys

Q: What is the type of this function?
float list -> float list -> float

Matrix product

let rec matprod arows brows =  
  let cols = transp brows in  
  map (fun row -> map (dotprod row) cols) arows
let rec matprod arows brows =
  let cols = transp brows in
  map (fun row -> map (dotprod row) cols) arows

Matrix Multiplication in OCaml
List Functionals for Predicates

let rec exists p = function
| [] -> false
| x::xs -> (p x) || (exists p xs)
val exists : ('a -> bool) -> ('a list -> bool) = <fun>

let rec filter p = function
| [] -> []
| x::xs ->
  if p x then
    x :: filter p xs
  else
    filter p xs
val filter : ('a -> bool) -> ('a list -> 'a list) = <fun>

(A predicate is a boolean-valued function.)
List Functionals for Predicates

Dual to exists:

```ml
let rec all p = function
  | [] -> true
  | x::xs -> (p x) && all p xs
```

val all : ('a -> bool) -> 'a list -> bool = <fun>

Example:

```ml
> exists (fun x -> x mod 2 = 0) [1; 2; 3];;  
- : bool = true

> filter (fun x -> x mod 2 = 0) [1; 2; 3];;  
- : int list = [2]

> all (fun x -> x mod 2 = 0) [1; 2; 3];;  
- : bool = false
```
Applications of the Predicate Functionals

let member y xs =
  exists (fun x -> x = y) xs;;

let inter xs ys =
  filter (fun x -> member x ys) xs;;

Testing whether two lists have no common elements

let disjoint xs ys =
  all (fun x -> all (fun y -> x <> y) ys) xs