Functions with more than one argument

```ocaml
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> <fun>
# let t = add_three 6 3 2;;
val t : int = 11
```

Again, first syntactic sugar for second

Functions with more than one argument

```ocaml
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> ... (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

Partial application of functions

```ocaml
let add_three x y z = x + y + z;;
```

```ocaml
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

- Partial application also called sectioning

Functions as arguments

```ocaml
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```
Tuples as Values

// r7 = \{c \rightarrow 4, test \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}
# let s = (5, "hi", 3.7);; val s : int * string * float = (5, "hi", 3.2)

Pattern Matching with Tuples

// r8 = \{s \rightarrow (5, "hi", 3.2), c \rightarrow 4, test \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}
# let (a,b,c) = s;; val a : int = 5
val b : string = "hi"
val c : float = 3.2

Functions on Tuples

# let plus_pair (n,m) = n + m;; val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;-: int = 7
# double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;-: int * int = (3, 3)
# double "hi";;-: string * string = ("hi", "hi")

Curried vs Uncurried

- Recall
val add_three : int -> int -> int -> int = <fun>
- How does it differ from
# let add_triple (u,v,w) = u + v + w;; val add_triple : int * int * int -> int = <fun>
  - add_three is \textit{curried};
  - add_triple is \textit{uncurried}

Nested Tuples

# (*Tuples can be nested *)
let d = ((1,4,62), ("bye",15), 73.95);;
val d : (int * int * int) * (string * int) * float = ((1, 4, 62), ("bye", 15), 73.95)
# (*Patterns can be nested *)
let (p,(st,_),_) = d;; val p : int * int * int = (1, 4, 62)
val st : string = "bye"
# Match Expressions

```ocaml
define triple_to_pair triple =
  match triple
  with (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);

let triple_to_pair : int * int * int -> int * int = <fun>
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

## Evaluate declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =

### Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration let \( x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \): \( \{ x \rightarrow v \} + \rho \)

### Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself, including primitive operators like + and =
Evaluating expressions in OCaml

- Evaluation uses an environment $\rho$
- A constant evaluates to itself, including primitive operators like $+$ and $=$
- To evaluate a variable, look it up in $\rho$: $\rho(v)$

To evaluate uses of $+$, $-$, etc, eval args, then do operation

Function expression evaluates to its closure

To evaluate a conditional expression: if $b$ then $e_1$ else $e_2$
  - Evaluate $b$ to a value $v$
  - If $v$ is True, evaluate $e_1$
  - If $v$ is False, evaluate $e_2$

To evaluate a local dec: let $x = e_1$ in $e_2$
  - Eval $e_1$ to $v$, then eval $e_2$ using $\{x \rightarrow v\} \cdot \rho$

To evaluate a tuple $(e_1, \ldots, e_n)$,
  - Evaluate each $e_i$ to $v_i$, right to left for Ocaml
  - Then make value $(v_1, \ldots, v_n)$
Evaluation of Application with Closures
- Given application expression \( fe \)
- In Ocaml, evaluate \( e \) to value \( v \)
- In environment \( \rho \), evaluate left term to closure, \( c = \langle x_1, \ldots, x_n \rangle \rightarrow b, \rho' \)
- \( x_1, \ldots, x_n \) variables in (first) argument
- \( v \) must have form \( (v_1, \ldots, v_n) \)
- Update the environment \( \rho' \) to
  \[ \rho'' = \{ x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n \} + \rho' \]
- Evaluate body \( b \) in environment \( \rho'' \)

Recursive Functions
- \# let rec factorial n =
  if \( n = 0 \) then 1 else \( n \times \text{factorial}(n - 1) \);
- val factorial : int -> int = <fun>
- \# factorial 5;;
  - : int = 120
- \# (* rec is needed for recursive function declarations *)

Recursion Example
Compute \( n^2 \) recursively using:
\[ n^2 = (2 \times n - 1) + (n - 1)^2 \]
- \# let rec nthsq n =
  match n with
  | 0 -> 0
  | n -> (2 \times n - 1) + nthsq(n - 1) ;;
- val nthsq : int -> int = <fun>
- \# nthsq 3;;
  - : int = 9
- Structure of recursion similar to inductive proof

Recursion and Induction
- \# let rec nthsq n = match n with
  | 0 -> 0
  | n -> (2 \times n - 1) + nthsq(n - 1) ;;
- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- if or match must contain base case
- Failure of these may cause failure of termination

Lists
- List can take one of two forms:
  - Empty list, written [ ]
  - Non-empty list, written \( x :: x s \)
    - \( x \) is head element, \( x s \) is tail list, :: called “cons”
  - Syntactic sugar: \([x]\) == \( x :: [\] \)
  - \([x_1; x_2; \ldots; x_n]\) == \( x_1 :: x_2 :: \ldots :: x_n :: [\] \)

Lists
- \# let fib5 = [8;5;3;2;1;1];;
  - val fib5 : int list = [8; 5; 3; 2; 1; 1]
- \# let fib6 = 13 :: fib5;;
  - val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
- \# (8::5::3::2::1::1::[]) = fib5;;
  - : bool = true
- \# fib5 @ fib6;;
  - : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
Lists are Homogeneous

# let bad_list = [1; 3.2; 7];;
Characters 19-22:
   let bad_list = [1; 3.2; 7];;
     ^^^
This expression has type float but is here used with type int

Question

- Which one of these lists is invalid?
  1. [2; 3; 4; 6]
  2. [2,3; 4,5; 6,7]
  3. [(2.3,4); (3.2,5); (6,7.2)]
  4. ["hi"; "there"]; ["wahcha"]; [ ]; ["doin"]

Answer

- Which one of these lists is invalid?

  1. [2; 3; 4; 6]
  2. [2,3; 4,5; 6,7]
  3. [(2.3,4); (3.2,5); (6,7.2)]
  4. ["hi"; "there"]; ["wahcha"]; [ ]; ["doin"]

- 3 is invalid because of last pair

Functions Over Lists

# let rec double_up list =
  match list
  with [] -> []  (* pattern before ->, expression after *)
    | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1]

# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]

Functions Over Lists

# let rec poor_rev list =
  match list
  with [] -> []
    | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1]

Structural Recursion

- Functions on recursive datatypes (eg lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function
Question: Length of list

Problem: write code for the length of the list
  - How to start?
  
  let rec length list =
Question: Length of list

- Problem: write code for the length of the list
- What result do we give when list is not empty?

```ocaml
let rec length list =
  match list with [] -> 0
  | (a :: bs) -> 1 + length bs
```

Structural Recursion: List Example

```
# let rec length list = match list 
  with [] -> 0 (* Nil case *)
  | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [] is base case
- Cons case recurses on component list bs

Same Length

- How can we efficiently answer if two lists have the same length?

```ocaml
let rec same_length list1 list2 =
  match list1 with [] -> true
  | (y::ys) -> false
  | (x::xs) ->
    (match list2 with [] -> false
     | (y::ys) -> same_length xys)
```

Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```ocaml
let rec doubleList list =
  match list with [] -> []
  | x :: xs -> (2 * x) :: doubleList xs
```

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2
Higher-Order Functions Over Lists

Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2.

```ocaml
let rec doubleList list =
  match list
  with 
    | [] -> []
    | x :: xs -> (2 * x) :: doubleList xs
```

Mapping Recursion

Can use the higher-order recursive map function instead of direct recursion.

```ocaml
# let rec map f list =
  match list
  with 
    | [] -> []
    | (H::T) -> (f H) :: (map f T);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x -1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Folding Recursion

Another common form “folds” an operation over the elements of the structure.

```ocaml
# let rec multList list =
  match list
  with 
    | [] -> 1
    | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 48
```

Higher-Order Functions Over Lists

Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2.

```ocaml
let rec doubleList list =
  match list
  with 
    | [] -> []
    | x :: xs -> (2 * x) :: doubleList xs
```

Mapping Recursion

Can use the higher-order recursive map function instead of direct recursion.

```ocaml
# let rec map f list =
  match list
  with 
    | [] -> []
    | (H::T) -> (f H) :: (map f T);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x -1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Folding Recursion

Another common form “folds” an operation over the elements of the structure.

```ocaml
# let rec multList list =
  match list
  with 
    | [] -> 1
    | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 48
```

Higher-Order Functions Over Lists

Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2.

```ocaml
let rec doubleList list =
  match list
  with 
    | [] -> []
    | x :: xs -> (2 * x) :: doubleList xs
```

Mapping Recursion

Can use the higher-order recursive map function instead of direct recursion.

```ocaml
# let rec map f list =
  match list
  with 
    | [] -> []
    | (H::T) -> (f H) :: (map f T);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x -1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Folding Recursion

Another common form “folds” an operation over the elements of the structure.

```ocaml
# let rec multList list =
  match list
  with 
    | [] -> 1
    | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 48
```

Higher-Order Functions Over Lists

Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2.

```ocaml
let rec doubleList list =
  match list
  with 
    | [] -> []
    | x :: xs -> (2 * x) :: doubleList xs
```

Mapping Recursion

Can use the higher-order recursive map function instead of direct recursion.

```ocaml
# let rec map f list =
  match list
  with 
    | [] -> []
    | (H::T) -> (f H) :: (map f T);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x -1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Folding Recursion

Another common form “folds” an operation over the elements of the structure.

```ocaml
# let rec multList list =
  match list
  with 
    | [] -> 1
    | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 48
```
Folding Recursion: Length Example

# let rec length list = match list
  with [ ] -> 0 (* Nil case *)
  | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>

# length [5; 4; 3; 2];;
- : int = 4

- Nil case [ ] is base case, 0 is the base value
- Cons case recurses on component list bs
- What do multList and length have in common?