Booleans (aka Truth Values)

```ocaml
# true;;
- : bool = true

# false;;
- : bool = false
```

```ocaml
// ρ₀ = {c → 4, test → 3.7, a → 1, b → 5}
# if b > a then 25 else 0;;
- : int = 25
```

Booleans and Short-Circuit Evaluation

```ocaml
# 3 > 1 && 4 > 6;;
- : bool = false

# 3 > 1 || 4 > 6;;
- : bool = true

# not (4 > 6);;
- : bool = true

# (print_string "Hi\n"; 3 > 1) || 4 > 6;;
Hi
- : bool = true

# 3 > 1 || (print_string "Bye\n"; 4 > 6);;
- : bool = true
```

Tuples as Values

```ocaml
// ρ₀ = {c → 4, a → 1, b → 5}
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

// ρ₁ = {s → (5, "hi", 3.2), a → 4, b → 5, c → 4}
# if b > a then (s, 25) else 0;
- : (int * string * float, int) = (5, "hi", 3.2, 25)
```

Pattern Matching with Tuples

```ocaml
// ρ₁ = {s → (5, "hi", 3.2), a → 1, b → 5, c → 4}

# let (a,b,c) = s;;
val a : int = 5
val b : string = "hi"
val c : float = 3.2

# let (a, _, _) = s;;
val a : int = 5

# let x = 2, 9.3;;
val x : int * float = (2, 9.3)
```

Nested Tuples

```ocaml
# (*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"
```
Functions on tuples

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

Save the Environment!

- A **closure** is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:
  
  `< (v1,...,vn) -> exp, ρ >`

- Where ρ is the environment in effect when the function is defined (for a simple function)

Closure for `plus_pair`

- Assume ρ_{plus_pair} was the environment just before `plus_pair` defined and recall
  
  ```ml
  # let plus_pair (n,m) = n + m;;
  val plus_pair : int * int -> int = <fun>
  ```

  - Closure for `fun (n,m) -> n + m:`
    
    `<(n,m) -> n + m, ρ_{plus_pair} >`

- Environment just after `plus_pair` defined:
  
  ```ml
  {plus_pair -> `<(n,m) -> n + m, ρ_{plus_pair} >} + ρ_{plus_pair}
  ```

Curried vs Uncurried

- Recall
  
  ```ml
  # let add_three u v w = u + v + w;;
  val add_three : int -> int -> int -> int = <fun>
  ```

  - How does it differ from
    
    ```ml
    # let add_triple (u,v,w) = u + v + w;;
    val add_triple : int * int * int -> int = <fun>
    ```

    - `add_three` is **curried**
    - `add_triple` is **uncurried**

Functions with more than one argument

```ml
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let t = add_three 6 3 2;;
val t : int = 11
# let add_triple (u,v,w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
# let add_triple =
  fun x ->
  (fun y ->
  (fun z -> x + y + z));;
val add_triple : int -> int -> int -> int = <fun>
```

```
Again, first syntactic sugar for second
```

Curried vs Uncurried

```ml
# add_three 6 3 2;;
- : int = 11
# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10: add_triple 5 4;;

```

```
This function is applied to too many arguments, maybe you forgot a `;`:
```

```ml
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

```
```
Partial application of functions

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

Recall: let plus_x = fun x => y + x

```ocaml
let x = 12
let plus_x = fun y => y + x
```

Closure for plus_x

- When plus_x was defined, had environment:
  \( \rho_{\text{plus}_x} = \{ \ldots, x \mapsto 12, \ldots \} \)
- Recall: let plus_x y = y + x
  is really let plus_x = fun y => y + x
- Closure for fun y => y + x:
  \( \langle y \mapsto y + x, \rho_{\text{plus}_x} \rangle \)
- Environment just after plus_x defined:
  \( \{ \text{plus}_x \mapsto \langle y \mapsto y + x, \rho_{\text{plus}_x} \rangle \} + \rho_{\text{plus}_x} \)

Evaluation

- Running Ocaml source:
  - Parse the program to detect each expression
  - Keep an internal environment at each time step
  - For each expression, interpret the program using the function **Eval**
  - Nice property of Ocaml: everything is a declaration or an expression!
- How does **Eval** (expression, environment) work:
  - Evaluation uses a starting environment \( \rho \)
  - Define the rules for evaluating declarations, constants, arithmetic expressions, function applications…

Evaluating Declarations

- Evaluation uses a starting environment \( \rho \)
- To evaluate a (simple) declaration let x = e
  - **Evaluate** expression e in \( \rho \) to value v
  - **Update** \( \rho \) with the mapping from x to v: \( \{ x \mapsto v \} + \rho \)

- **Update**: \( \rho_1 + \rho_2 \) has all the bindings in \( \rho_1 \) and all those in \( \rho_2 \) that are not rebound in \( \rho_1 \)

  \[
  \{ x \mapsto 2, y \mapsto 3, a \mapsto "hi" \} + \{ y \mapsto 100, b \mapsto 6 \} = \{ x \mapsto 2, y \mapsto 3, a \mapsto "hi" \} + \{ b \mapsto 6 \}
  \]

- It is not commutative!

Warm-up: we evaluate this case:

```ocaml
\( \rho = \{ x \mapsto 2 \} \)
let y = 2*x+1;;
\( \rho' = \{ x \mapsto 2; y \mapsto 5 \} \)
Evaluating Expressions

- Evaluation uses an environment \( \rho \)

- A constant evaluates to itself

- To evaluate an variable, look it up in \( \rho \), i.e., use \( \rho(v) \)

- To evaluate uses of \(+\), \(-\), etc., first eval the arguments, then do operation

- To evaluate a local declaration: let \( x = e_1 \) in \( e_2 \)
  - Evaluate \( e_1 \) to \( v \), evaluate \( e_2 \) using \( \{ x \mapsto v \} + \rho \)

Function application \((f x)\) -- see next slide

Evaluation of Application with Closures

- Function defined as: let \( f(x_1, \ldots, x_n) = \text{body} \)

- Function application: \( f(e_1, \ldots, e_n) \)

- Evaluation uses the function App(Closure, Value):
  - In environment \( \rho \), evaluate the left term to closure, \( c = \langle (x_1, \ldots, x_n) \mapsto \text{body}, \rho \rangle \)
  - Evaluate the arguments in the application \( e_1, \ldots, e_n \) to their values \( v_1, \ldots, v_n \) in the environment \( \rho \)
  - Update the environment \( \rho \) to \( \rho' = \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \} + \rho \)
  - Evaluate the function body (body) in environment \( \rho' \)

Evaluation of Application of plus_x;;

- Have environment:
  \[
  \rho = \{ \text{plus}_x \mapsto \lambda y \to y + x, \rho_{\text{plus}_x} >, \ldots, y \to 3, \ldots \}
  \]

  where \( \rho_{\text{plus}_x} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)

  - Eval \((\text{plus}_x y, \rho)\) rewrites to
  - App \((\text{Eval(plus}_x, \rho), \text{Eval(y, \rho)})\) rewrites to
  - App \((\langle y \to y + x, \rho_{\text{plus}_x} >, 3 \rangle\) rewrites to
  - Eval \((y + x, (y \to 3) + \rho_{\text{plus}_x})\) rewrites to
  - Eval \((3 + 12, \rho_{\text{plus}_x}) = 15\)

Evaluation of Application of plus_pair

- Assume environment
  \[
  \rho = \{ x \to 3, \ldots, \text{plus}_\text{pair} \mapsto \lambda (n,m) \to n + m, \rho_{\text{plus}_\text{pair}} > \}
  \]

  where \( \rho_{\text{plus}_\text{pair}} = \{ n \mapsto 12, \ldots, m \mapsto 24, \ldots \} \)

  - Eval \((\text{plus}_\text{pair} (4, x), \rho)\) =
  - App \((\text{Eval(plus}_\text{pair}, \rho), \text{Eval((4,x), \rho)}\) =
  - App \((\langle n,m \mapsto n + m, \rho_{\text{plus}_\text{pair}} >, (4, 3) \rangle\) =
  - Eval \((n + m, (n \to 4, m \to 3) + \rho_{\text{plus}_\text{pair}}) =
  - Eval \((4 + 3, (n \to 4, m \to 3) + \rho_{\text{plus}_\text{pair}}) = 7\)

Closure question

- If we start in an empty environment, and we execute:
  
  let \( f = \text{fun n -> n + 5;}; \)
  
  (* 0 *)
  
  let \( \text{pair}_\text{map} \text{ g}(n,m) = (g n, g m);;; \)
  
  let \( f = \text{pair}_\text{map} f; ;; \)
  
  let \( a = f (4,6);;; \)

  What is the environment at (* 0 *)?

Answer

\( \rho_{\text{start}} = \{ \} \)

let \( f = \text{fun n -> n + 5;}; \)

\( \rho_0 = \{ f \to \langle n \to n + 5, \{ \} \rangle \}\)
Closure question

If we start in an empty environment, and we execute:

\[
\begin{align*}
\text{let } f &= \text{fun } n \Rightarrow n + 5; \\
\text{let } \text{pair_map } g (n,m) &= (g \, n, \, g \, m); \\
(*) \, 1 \,* \\
\text{let } f &= \text{pair_map } f; \\
\text{let } a &= f \, (4,6); \\
\end{align*}
\]

What is the environment at (*) 1 *? 

Answer

\[
\begin{align*}
\rho_0 &= \{ f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \\
\text{let } \text{pair_map } g (n,m) &= (g \, n, \, g \, m); \\
\rho_1 &= \\
&\{ \\
&f \mapsto \langle n \mapsto n + 5, \{ \} \rangle, \\
&\text{pair_map} \mapsto \\
&\langle g \mapsto \langle \text{fun} \, (n,m) \mapsto (g \, n, \, g \, m) \rangle, \\
&\{f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \rangle \\
&\} \\
\end{align*}
\]

Closure question

If we start in an empty environment, and we execute:

\[
\begin{align*}
\text{let } f &= \text{fun } n \Rightarrow n + 5; \\
\text{let } \text{pair_map } g (n,m) &= (g \, n, \, g \, m); \\
\text{let } f &= \text{pair_map } f; \\
(*) \, 2 \,* \\
\text{let } a &= f \, (4,6); \\
\end{align*}
\]

What is the environment at (*) 2 *? 

Evaluate \text{pair_map } f

\[
\begin{align*}
\rho_0 &= \{ f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \\
\rho_1 &= \{ f \mapsto \langle n \mapsto n + 5, \{ \} \rangle, \\
&\text{pair_map} \mapsto \\
&\langle g \mapsto \langle \text{fun} \, (n,m) \mapsto (g \, n, \, g \, m) \rangle, \\
&\{f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \rangle \\
\text{let } f &= \text{pair_map } f; \\
\end{align*}
\]

\[
\text{Eval}(\text{pair_map } f, \rho_1) = 
\]

Evaluate \text{pair_map } f

\[
\begin{align*}
\rho_0 &= \{ f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \\
\rho_1 &= \{ f \mapsto \langle n \mapsto n + 5, \{ \} \rangle, \\
&\text{pair_map} \mapsto \\
&\langle g \mapsto \langle \text{fun} \, (n,m) \mapsto (g \, n, \, g \, m) \rangle, \\
&\{f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \rangle \\
\text{let } f &= \text{pair_map } f; \\
\text{Eval}(\text{pair_map } f, \rho_1) = 
\end{align*}
\]

\[
\text{App} \langle g \mapsto \langle \text{fun} \, (n,m) \mapsto (g \, n, \, g \, m) \rangle, \rho_0 \rangle, \langle n \mapsto n + 5, \{ \} \rangle = 
\]
Evaluate \( \text{pair\_map} \ f \)

\[
\rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}
\]
\[
\rho_1 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle ,
\quad \text{pair\_map} \rightarrow
\quad \langle g \rightarrow \langle \text{fun}(n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \rangle \rangle \}
\]
\]
\[
\text{let } f = \text{pair\_map} \ f ;
\]

\[
\text{Eval}(\text{pair\_map} \ f, \rho_1) =
\]

\[
\text{App}(\langle g \rightarrow \langle \text{fun}(n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \rangle \rangle, \\
\langle \langle n \rightarrow n + 5, \{ \} \rangle \rangle \rangle
\]

Answer

\[
\rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}
\]
\[
\rho_1 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle ,
\quad \text{pair\_map} \rightarrow
\quad \langle g \rightarrow \langle \text{fun}(n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \rangle \rangle \}
\]
\]
\[
\text{let } f = \text{pair\_map} \ f ;
\]

\[
\rho_2 = \{ f \rightarrow \langle (n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle \},
\quad \text{pair\_map} \rightarrow \langle g \rightarrow \langle \text{fun}(n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle \rangle \rangle, \\
\quad \{ \langle n \rightarrow n + 5, \{ \} \rangle \rangle \}
\]
\]
\[
\text{let } a = f (4,6) ; ;
\]

**Closure question**

- If we start in an empty environment, and we execute:
  - let \( f = \text{fun} \rightarrow n + 5 ; ; \)
  - let \( \text{pair\_map} \ g \ (n,m) = (g \ n, g \ m) ; ; \)
  - let \( f = \text{pair\_map} \ f ; ; \)
  - let \( a = f \ (4,6) ; ; \)

What is the environment at \(* 3 *\)?

**Final Evaluation?**

\[
\rho_2 = \{ f \rightarrow \langle (n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle \},
\quad \text{pair\_map} \rightarrow \langle g \rightarrow \langle \text{fun}(n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle \rangle, \\
\quad \{ \langle n \rightarrow n + 5, \{ \} \rangle \rangle \}
\]
\]
\[
\text{let } a = f (4,6) ; ;
\]

**Evaluate \( f (4,6) ; ; \)**

\[
\rho_2 = \{ f \rightarrow \langle (n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle \},
\quad \text{pair\_map} \rightarrow \langle g \rightarrow \langle \text{fun}(n,m) \rightarrow (g \ n, g \ m) \rangle, \\
\quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle \rangle, \\
\quad \{ \langle n \rightarrow n + 5, \{ \} \rangle \rangle \}
\]
\]
\[
\text{let } a = f (4,6) ; ;
\]

\[
\text{Eval}(f (4,6), \rho_2) =
\]

\[
(4,6) =
\]

9/10/2017
Evaluate \( f(4,6) \):

\[
\text{App}(\langle n, m \rangle \rightarrow (g \ n, g \ m), \{ g \mapsto \langle n \rightarrow n + 5, \{ \rangle, f \mapsto \langle n \rightarrow n + 5, \{ \rangle \} \}) = \]
\[
\text{Eval}(\langle n \mapsto n + 5, \{ \rangle, 4), \text{App}(\langle n \mapsto n + 5, \{ \rangle, 6)) =}
\]

9/10/2017 37

9/10/2017 38

Functions as arguments

# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>

# let g = thrice plus_two;; (* plus_two x is x+2 *)
val g : int -> int = <fun>

# g 4;;
- : int = 10

# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"

9/10/2017 39

Higher Order Functions

- A function is higher-order if it takes a function as an argument or returns one as a result

- Example:

  # let compose f g = fun x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

  - The type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b is a higher order type because of
    ('a -> 'b) and ('c -> 'a) and -> 'c -> 'b

9/10/2017 40

Thrice

- Recall:

  # let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>

- How do you write thrice with compose?

  # let thrice f = compose f (compose f f);
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>

9/10/2017 41

Lambda Lifting

- You must remember the rules for evaluation when you use partial application

  # let add_two = (+) (print_string "test\n"; 2);
test
val add_two : int -> int = <fun>

  # let add2 = (* lambda lifted *)
  fun x -> (+) (print_string "test\n"; 2) x;
val add2 : int -> int = <fun>

9/10/2017 42
Lambda Lifting

```ocaml
# thrice add_two 5;;
. : int = 11

# thrice add2 5;;
test
test
test
. : int = 11
```

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

Match Expressions

```ocaml
# let triple_to_pair triple =

match triple with
 (0, x, y) -> (x, y)
| (x, 0, y) -> (x, y)
| (x, y, _) -> (x, y);;
```

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

```ocaml
val triple_to_pair : int * int * int -> int * int = <fun>
```

Recursive Functions

```ocaml
# let rec factorial n =
    if n = 0 then 1
    else n * factorial (n - 1);;
val factorial : int -> int = <fun>

# factorial 5;;
- : int = 120
```

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

```ocaml
# (* rec is needed for recursive function declarations *)
```

Recursion Example

```ocaml
# let rec nthsq n =
    match n with
    0 -> 0 (*Base case!*)
| n -> (2 * n - 1) + nthsq (n - 1);;

val nthsq : int -> int = <fun>
```

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

```ocaml
# nthsq 3;;
- : int = 9
```

- Structure of recursion similar to inductive proof

Recursion and Induction

```ocaml
# let rec nthsq n =
    match n with
    0 -> 0 (*Base case!*)
| n -> (2 * 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 selecting base case will cause non-termination
  - But the program will crash because it exhausts the stack!

Lists

- First example of a recursive datatype (aka algebraic datatype)
- Unlike tuples, lists are homogeneous in type (all elements same type)
Lists

- List can take one of two forms:
  - Empty list, written \([\ ]\)
  - Non-empty list, written \(x :: xs\)
    - \(x\) is head element,
    - \(xs\) is tail list, :: called “cons”
- How we typically write them (syntactic sugar):
  - \([x]\) == \(x :: [\ ]\)
  - \([x_1; x_2; ...; x_n]\) == \(x_1 :: x_2 :: ... :: x_n :: [\ ]\)

```
# 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;[ ]) = fib5;;
val true : bool
# fib5 @ fib6;;
val true : 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"; \[ "wahcha"; [ ]; "doin"]\]"

```
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>

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

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

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

# poor_rev silly;;
val true : string list = ["there"; "there"; "hi"; "hi"]
```

Question: Length of list

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

9/10/2017

Question: Length of list

- Problem: write code for the length of the list
  - How to start?
  let rec length l =
    match l with

9/10/2017

Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?
  let rec length l =
    match l with [] -> 0
    | (a :: bs) ->

9/10/2017

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when l is empty?
  let rec length l =
    match l with [] -> 0
    | (a :: bs) ->

9/10/2017
Question: Length of list

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

```ocaml
let rec length l =
  match l with
  | []       -> 0          
  | (a :: bs) -> 1 + length 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 
  | (x :: xs) -> same_length xs list2 

let rec map f list =
  match list with
  | []       -> []          
  | (h :: t) -> (f h) :: (map f t);

let rec fold_left f a list =
  match list with
  | []       -> a          
  | (x :: xs) -> fold_left f (f a x) xs;

let rec fold_right f list b =
  match list with
  | []       -> b          
  | (x :: xs) -> f x (fold_right f xs b);
```

Functions Over Lists

```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]

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

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

# fold_left
  (fun () -> print_string)
  ()
  ();

# fold_right
  (fun s -> fun () -> print_string s)
  "hi"; "there";
val therehi : unit = ()
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

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion is a form of Structural Recursion
  - In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
  - Wait until whole structure has been traversed to start building answer

Encoding Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with
  | [] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Mapping Recursion

- One common form of structural recursion applies a function to each element in the structure

```ocaml
# let rec doubleList list = match list with
  | [] -> [] | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Mapping Recursion

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

```ocaml
# let doubleList list =  
    List.map (fun x -> 2 * x) list;;  
val doubleList : int list -> int list = <fun>

# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

- Same function, but no recursion

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;4;6];;
- : int = 48
```

- Computes \(2 \times (4 \times (6 \times 1))\)

How long will it take?

Common big-O times:

- Constant time \(O(1)\)
  - input size doesn’t matter
- Linear time \(O(n)\)
  - \(2x\) input size \(\Rightarrow 2x\) time
- Quadratic time \(O(n^2)\)
  - \(3x\) input size \(\Rightarrow 9x\) time
- Exponential time \(O(2^n)\)
  - Input size \(n+1\) \(\Rightarrow 2^n\) time

Linear Time

- Expect most list operations to take linear time \(O(n)\)
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: `multList`, `append`
- Integer example: `factorial`

Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

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

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

### Exponential running time

```ocaml
# let rec naiveFib n = match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> naiveFib (n-1) + naiveFib (n-2);
val naiveFib : int -> int = <fun>
```

An Important Optimization

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if \( f \) calls \( g \) and \( g \) calls \( h \), but calling \( h \) is the last thing \( g \) does (a tail call)?
- Then \( h \) can return directly to \( f \) instead of \( g \)

### An Important Optimization

![Diagram of normal and tail call]

Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls
- Tail recursion generally requires extra “accumulator” arguments to pass partial results
- May require an auxiliary function

### Tail Recursion - Example

```ocaml
# let rec rev_aux list revlist = match list with
  | [] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list -> 'a list = <fun>
```

```ocaml
# let rev list = rev_aux list [];;
val rev : 'a list -> 'a list = <fun>
```

- What is its running time?
Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with
  | [] -> 0 | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>

# sumlist [2;3;4];;
. : int = 9

# let rec prodlist list = match list with
  | [] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>

# prodlist [2;3;4];;
. : int = 24
```

Folding - Forward Recursion

```ocaml
# let sumlist list = fold_right (+) list 0;;
val sumlist : int list -> int = <fun>

# sumlist [2;3;4];;
. : int = 9

# let prodlist list = fold_right ( * ) list 1;;
val prodlist : int list -> int = <fun>

# prodlist [2;3;4];;
. : int = 24
```

Folding - Tail Recursion

```ocaml
# let rev list =
  fold_left (fun l -> fun x -> x :: l) //comb op
  [] //accumulator cell
  list
```

Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
- Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition