Tuples as Values

// \( \rho_0 = \{ c \rightarrow 4, a \rightarrow 1, b \rightarrow 5 \} \)
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

// \( \rho = \{ s \rightarrow (5, "hi", 3.2), c \rightarrow 4, a \rightarrow 1, b \rightarrow 5 \} \)
Pattern Matching with Tuples

// \( \rho = \{ s \rightarrow (5, "hi", 3.2), a \rightarrow 1, b \rightarrow 5, c \rightarrow 4 \} \)

# let (a,b,c) = s;;  (* (a,b,c) is a pattern *)
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;;  (* tuples don't require parens in Ocaml *)
val x : int * float = (2, 9.3)
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"

9/6/2018
Functions on tuples

# 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")
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:

\[ \langle (v_1, \ldots, v_n) \rightarrow \text{exp}, \ \rho \rangle \]

Where \( \rho \) is the environment in effect when the function is defined (for a simple function)
Closure for `plus_pair`

- Assume $\rho_{plus\_pair}$ was the environment just before `plus_pair` defined and recall
  - `let plus_pair (n,m) = n + m;;`

- Closure for `fun (n,m) -> n + m:`
  $$\langle (n,m) \rightarrow n + m, \rho_{plus\_pair} \rangle$$

- Environment just after `plus_pair` defined:
  $$\{ plus\_pair \rightarrow \langle (n,m) \rightarrow n + m, \rho_{plus\_pair} \rangle \} + \rho_{plus\_pair}$$

Like set union! (but subtle differences, see slide 17)
Functions with more than one argument

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

Again, first syntactic sugar for second
Curried vs Uncurried

- Recall

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

- How does it differ from

```ocaml
# 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**
Curried vs Uncurried

# 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 `;'

# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
Partial application of functions

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

# let h = add_three 5 4;;
val h : int -> int = <fun>

# h 3;;
- : int = 12

# h 7;;
- : int = 16

Partial application also called *sectioning*
Recall: let plus\_x = fun y -> y + x

let x = 12

X \rightarrow 12

let plus\_x = fun y -> y + x

let x = 7

\[ x \rightarrow 12 \]

\[ y \rightarrow y + x \]

\[ y \rightarrow y + x \]

\[ x \rightarrow 12 \]

\[ x \rightarrow 7 \]
Closure for `plus_x`

- When `plus_x` was defined, had environment:
  
  \[ \rho_{plus_x} = \{\ldots, x \mapsto 12, \ldots\} \]

- Recall: \texttt{let plus_x y = y + x}
  
  is really \texttt{let plus_x = fun y -> y + x}

- Closure for \texttt{fun y -> y + x}:
  
  \[ <y \mapsto y + x, \rho_{plus_x}> \]

- Environment just after `plus_x` defined:
  
  \[ \{plus_x \mapsto <y \mapsto y + x, \rho_{plus_x}>, \rho_{plus_x}\} + \rho_{plus_x} \]
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 (mathematical) function \texttt{Eval}
  - Nice property of Ocaml: \textit{everything is a declaration or an expression}!

- How does \texttt{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 $\text{let } x = e$
    - **Evaluate** expression $e$ in $\rho$ to value $v$
    - **Update** $\rho$ with the mapping from $x$ to $v$: $\{x \rightarrow v\} + \rho$

**Definition of $+$ on environments!**

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

$$\{x \rightarrow 2, \ y \rightarrow 3, \ a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, \ b \rightarrow 6\} = \{x \rightarrow 2, \ y \rightarrow 3, \ a \rightarrow \text{“hi”}, \ b \rightarrow 6\}$$

It is not commutative!
Evaluating Declarations

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

Warm-up: we evaluate this case:

$$\rho = \{ x \rightarrow 2 \}$$

let $y = 2*x+1;$

$$\rho' = \{ x \rightarrow 2; \ y \rightarrow 5 \}$$
Evaluating Expressions (Rules)

- Evaluation uses an environment $\rho$
- A constant evaluates to itself
- To evaluate a variable $x$, look it up in $\rho$ i.e., use $\rho(x)$
- To evaluate tuples, evaluate each tuple element
- To evaluate uses of $+$, $-$, etc, first eval the arguments, then do the operation
- To evaluate a local declaration: $\text{let } x = e_1 \text{ in } e_2$
  - Evaluate $e_1$ to $v$, evaluate $e_2$ using $\{x \rightarrow v\} + \rho$
- Function application $(f \ x)$ -- see next slide
Evaluation of Function Application with Closures

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

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

Let us define \( \text{Eval}( f (e_1, \ldots, e_n), \rho) \):

- In the environment \( \rho \), evaluate the left term (\( f \)) to closure, i.e.,
  \( c = <(x_1, \ldots, x_n) \rightarrow \text{body}, \rho^* > \)

- Evaluate the arguments in the application \( e_1 \ldots e_n \) to their values \( v_1, \ldots, v_n \) in the environment \( \rho \)

- Call helper function \( \text{App}(\text{Closure, Value}) \) to evaluate the function body (\( \text{body} \)) in the environment \( \rho^* \)
  - Conjoin the mapping of the arguments to values with the environment \( \rho^* \)
    \[ \rho' = \{ x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n \} + \rho^* \]
  - The App then calls Eval again for the expressions in body in the env. \( \rho' \)
Evaluation of Application of `plus_x`;

- Have environment:
  \[
  \rho = \{\text{`plus_x`} \rightarrow \langle y \rightarrow y + x, \rho_{\text{`plus_x`}} \rangle, \ldots, y \rightarrow 3, \ldots\}\]

where \( \rho_{\text{`plus_x`}} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\} \)

- \(\text{Eval (`plus_x` } y, \rho)\) rewrites to
- \(\text{App (Eval(`plus_x`} , \rho), \text{Eval}(y, \rho))\) rewrites to
- \(\text{App (}\langle y \rightarrow y + x, \rho_{\text{`plus_x`}} \rangle, 3\rangle\) rewrites to
- \(\text{Eval (} y + x, \{y \rightarrow 3\} + \rho_{\text{`plus_x`}}\)\) rewrites to
- \(\text{Eval (} 3 + 12 \ , \rho_{\text{`plus_x`}}\) = 15\)
Evaluation of Application of `plus_pair`

- **Assume environment**

  \[ \rho = \{ x \mapsto 3, \ldots, \} \]

  \[ \text{plus_pair} \mapsto \langle n,m \mapsto n + m, \rho_{\text{plus_pair}} \rangle \] + \rho_{\text{plus_pair}} \]

- **Eval** \((\text{plus_pair} (4,x), \rho)\) =

- **App** \(\text{Eval (plus_pair, } \rho)\), \(\text{Eval } ((4,x), \rho)\) =

- **App** \(\langle n,m \mapsto n + m, \rho_{\text{plus_pair}} \rangle, (4,3)\) =

- **Eval** \(n + m, \{ n \mapsto 4, m \mapsto 3 \} + \rho_{\text{plus_pair}} \) =

- **Eval** \(4 + 3, \{ n \mapsto 4, m \mapsto 3 \} + \rho_{\text{plus_pair}} \) = 7
Closure question

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

```ml
let f = fun n -> n + 5;;
(* 0 *)
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
let a = f (4,6);;
```

What is the environment at (* 0 *?)?
$\rho_{\text{start}} = \{\}$

let $f = \text{fun } n \rightarrow n + 5;;$

$\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{\}\rangle\}$
Closure question

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

```ocaml
let f = fun n -> n + 5;;
let pair_map g (n,m) = (g n, g m);;
(* 1 *)
let f = pair_map f;;
let a = f (4,6);;
```

What is the environment at (* 1 *?)?
Answer

\[ \rho_0 = \{ f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \]

``` Ocaml
let pair_map g (n,m) = (g n, g m);
```

\[ \rho_1 = \{ \]

``` Ocaml
   f \mapsto \langle n \mapsto n + 5, \{ \} \rangle, 
   pair_map \mapsto 
     \langle g \mapsto (fun (n,m) -> (g n, g m)), 
         \{ f \mapsto \langle n \mapsto n + 5, \{ \} \rangle \} \rangle 
   \}
```

9/6/2018
Closure question

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

```ocaml
let f = fun n -> n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
(* 2 *)
let a = f (4,6);;
```

What is the environment at (* 2 *?)?
Evaluate `pair_map f`

\[ \rho_0 = \{ f \mapsto <n \mapsto n + 5, \{ \} > > \} \]

\[ \rho_1 = \{ f \mapsto <n \mapsto n + 5, \{ \} > > , \]

\[ \text{pair_map} \rightarrow \]

\[ <g \mapsto \text{(fun (n,m) -> (g n, g m))}, \]

\[ \{ f \mapsto <n \mapsto n + 5, \{ \} > > > \} \]

\[ \text{let } f = \text{pair_map } f ;; \]
Evaluate \texttt{pair\_map \ f} \\

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

let \ f = \text{pair\_map} \ f;; \\

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

\[
\rho_0 = \{ f \mapsto <n \mapsto n + 5, \{ \}>, \text{pair\_map} \mapsto <g \mapsto \text{fun (n,m) \rightarrow (g n, g m)}, \{f \mapsto <n \mapsto n + 5, \{ \}>>}\}
\]

\[
\rho_1 = \{ f \mapsto <n \mapsto n + 5, \{ \}>, \text{pair\_map} \mapsto <g \mapsto \text{fun (n,m) \rightarrow (g n, g m)}, \{f \mapsto <n \mapsto n + 5, \{ \}>>}\}
\]

let \( f = \text{pair\_map \ f};; \)

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

\[
\text{App (<g \rightarrow \text{fun (n,m) \rightarrow (g n, g m)}, \rho_0>, <n \mapsto n + 5, \{ \}>) =}
\]
Evaluate \texttt{pair\_map f}

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

\texttt{let f = pair\_map f;;}

\texttt{Eval(pair\_map f, \rho_1) =}
\texttt{App (\langle g \rightarrow \text{fun} \ (n,m) \rightarrow (g \ n, g \ m), \rho_0 \rangle, \langle n \rightarrow n + 5, \{ \} \rangle) =}

\texttt{Eval(fun \ (n,m)\rightarrow(g \ n, g \ m), \{ g\rightarrow\langle n\rightarrow n + 5, \{ \} \rangle \} + \rho_0) =}
\texttt{\langle (n,m) \rightarrow (g \ n, g \ m), \{ g\rightarrow\langle n\rightarrow n + 5, \{ \} \rangle \} + \rho_0 \rangle =}
\texttt{\langle (n,m) \rightarrow (g \ n, g \ m), \{ g\rightarrow\langle n\rightarrow n + 5, \{ \} \rangle, f\rightarrow\langle n\rightarrow n + 5, \{ \} \rangle \} \rangle}

9/6/2018
Answer

\[ \rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \]
\[ \rho_1 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle, \]
\[ \text{pair_map} \rightarrow \]
\[ \quad \langle g \rightarrow (\text{fun} (n,m) \rightarrow (g \ n, g \ m)), \]
\[ \quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \} \} \}
\]
\[ \text{let } f = \text{pair_map } f;; \]
\[ \rho_2 = \{ f \rightarrow \langle (n,m) \rightarrow (g \ n, g \ m), \]
\[ \quad \{ g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle, \]
\[ \quad f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \} \} \}, \]
\[ \text{pair_map} \rightarrow \langle g \rightarrow \text{fun} (n,m) \rightarrow (g \ n, g \ m), \]
\[ \quad \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \}
\[ \quad > \]
\[ \} \]
Closure question

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

```ocaml
let f = fun n -> n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
let a = f (4,6);;
(* 3 *)
```

What is the environment at (* 3 * )?
Final Evaluation?

\( \rho_2 = \{ f \rightarrow (n,m) \rightarrow (g \ n, g \ m), \)

\[
\{ g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle, \\
  f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \}\}, \\
\]

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

let a = f (4,6);

9/6/2018
Evaluate $f(4,6)$;

\[
\rho_2 = \{ f \to <(n,m) \to (g \ n, \ g \ m), \\
\quad \{ g \to <n \to n + 5, \{ \}\}, \\
\quad f \to <n \to n + 5, \{ \}\}, \}
\]

\[
\text{pair}_\text{map} \to <g \to \text{fun}(n,m) \to (g \ n, \ g \ m), \\
\quad \{ f \to <n \to n + 5, \{ \}\} \\
\]

let a = f (4,6);

Eval(f (4,6), \rho_2) =
Evaluate $f(4,6)$;

$$
\rho_2 = \{ f \to <(n,m) \to (g\ n, \ g\ m),
\begin{array}{l}
\{ g \to <n \to n + 5, \{ \}\},
\{ f \to <n \to n + 5, \{ \}\}\}\},
\end{array}
\}
$$

let a = $f(4,6)$;

\[
\text{Eval}(f(4,6), \rho_2) = \\
\text{App}(<(n,m) \to (g\ n, \ g\ m), \{ g \to <n \to n + 5, \{ \}\},
\begin{array}{l}
\{ f \to <n \to n + 5, \{ \}\}\}\},
\end{array}
\}
(4,6)) =
\]
Evaluate $f(4,6)$;

$$\text{App(<(n,m) \rightarrow (g n, g m), \{g \rightarrow <n \rightarrow n + 5, \{\}\>, f \rightarrow <n \rightarrow n + 5, \{\}>\}>>,
        (4,6)) =}$$

$$\text{Eval((g n, g m), \{n \rightarrow 4, m \rightarrow 6\} + \{g \rightarrow <n \rightarrow n + 5, \{\}\>, f \rightarrow <n \rightarrow n + 5, \{\}>\} =}$$

$$(\text{App(<n \rightarrow n + 5, \{\}>>, 4), \text{App (<n \rightarrow n + 5, \{\}>>, 6)) =}$$
Evaluate \( f(4, 6) \);,

\[
\text{App}(\langle n \rightarrow n + 5, \{ \}\rangle, 4), \\
\text{App}(\langle n \rightarrow n + 5, \{ \}\rangle, 6)) = \\
(\text{Eval}(n + 5, \{n \rightarrow 4\} + \{ \})), \\
\text{Eval}(n + 5, \{n \rightarrow 6\} + \{ \}))) = \\
(\text{Eval}(4 + 5, \{n \rightarrow 4\} + \{ \})), \\
\text{Eval}(6 + 5, \{n \rightarrow 6\} + \{ \}))) = (9, 11)
\]

Finally:

\( \rho_3 = \{a \rightarrow (9, 11)\} + \rho_2 \)
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!"
A function is *higher-order* if it takes a function as an argument or returns one as a result.

**Example:**

```ocaml
# 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`
Thrice

- Recall:

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

- How do you write thrice with compose?

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

# (+)
- : int -> int -> int = <fun>

# let add_two = (+) (print_string "test\n"; 2);;

# let add2 = (* lambda lifted *)
  fun x -> (+) (print_string "test\n"; 2) x;;
Lambda Lifting

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

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

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

# 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
Reminder: Pattern Matching with Tuples

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

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

# (*Patterns can be nested *)
# let (p, (st,_), _) = d;;
# (* _ matches all, binds nothing *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
Match Expressions

# let triple_to_pair triple =

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

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

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

Compute $n^2$ recursively using:

\[ n^2 = (2 \times n - 1) + (n - 1)^2 \]

```plaintext
let rec nthsq n =         (* rec for recursion *)
    match n with    (* pattern matching for cases *)
      0 -> 0       (* base case *)
    | n -> (2 * n -1) (* recursive case *)
      + nthsq (n -1);; (* recursive call *)

val nthsq : int -> int = <fun>

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

Structure of recursion similar to inductive proof
Recursion and Induction

```ml
# 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 the 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):

- \([x]\) == \(x :: [\ ]\)
- \([x_1; x_2; \ldots; x_n]\) == \(x_1 :: x_2 :: \ldots :: x_n :: [\ ]\)
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”]]

3 is invalid because of the 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]
Functions Over Lists

# 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;;
- : 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 =
Question: Length of list

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

```ml
let rec length l =
  match l with
```
Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```ocaml
let rec length l =
  match l with
```
Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```ocaml
let rec length l =
  match l with [] ->
   | (a :: bs) ->
```
Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when \( l \) is empty?

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

9/6/2018
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) ->
```

9/6/2018
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
```
How can we efficiently answer if two lists have the same length?

**Tactics:**
- First list is empty: then true if second list is empty else false
- First list in not empty: then if second list empty return false, or otherwise compare whether the sublists (after the first element) have the same length
Same Length

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

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

9/6/2018
# 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]
Iterating over lists

# 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>

# fold_left
  (fun () -> print_string)
  ()
  ["hi"; "there"];;
hithere- : unit = ()
Iterating over lists

# 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_right
  (fun s -> fun () -> print_string s)
  ["hi"; "there"]
  ();;
therehi- : unit = ()
Structural Recursion

- Functions on recursive datatypes (e.g., 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.
 Structural Recursion : List Example

```ocaml
# let rec length list =
  match list with
    [] -> 0                      (* Nil case *)
  | x :: xs -> 1 + length xs;;  (* Cons case *)

val length : 'a list -> int = <fun>
```

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

- Nil case `[]` is base case
- Cons case recurses on component list `xs`
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
Forward Recursion: Examples

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

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

# 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]

# let append_alt list1 list2 =
    fold_right (fun x y -> x :: y) list1 list2;;
val append_alt : 'a list -> 'a list -> 'a list = <fun>
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

```
# 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 \((2 \times (4 \times (6 \times 1)))\)
Folding Recursion

- `multList` folds to the right
- Same as:

```ocaml
# let multList list =
  List.fold_right
    (fun x -> fun p -> x * p)
  list 1;;
val multList : int list -> int = <fun>

# multList [2;4;6];;
- : int = 48
```
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 $\Rightarrow$ 2x 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

# 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)?
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 \( g \).
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

```hs
# 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>

# 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

# 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>
fold_left f a [x₁; x₂;...;xₙ] = f(...(f (f a x₁) x₂)...xₙ)

# 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_right f [x₁; x₂;...;xₙ] b = f x₁(f x₂ (...(f xₙ b)...))
Folding - Forward Recursion

# 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

- # 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