Thinking Inductively

COS 326
David Walker
Princeton University
Assignment 1 due at 11:59pm tonight!

Program style guide:

Read notes:
- functional basics, type-checking, typed programming
- thinking inductively (today)
- Real World OCaml Chapter 2, 3 (optional)

For Windows users:
- I pinned some install instructions to the top of the Piazza feed
- They look relatively easy to follow and have the side effect of installing bash, which will be generally useful for you in the future
Options

A value $v$ has type $t$ option if it is either:

– the value `None`, or
– a value `Some v'`, and $v'$ has type $t$

Options can signal there is no useful result to the computation

Example: we look up a value in a hash table using a key.

– If the key is present, return `Some v` where $v$ is the associated value
– If the key is not present, we return `None`
Slope between two points

```
type point = float * float

let slope (p1:point) (p2:point) : float =
```

![Diagram of slope between two points]

(a) and (b) are the horizontal and vertical distances, respectively.

(c) is the hypotenuse of the right triangle formed by the points.

The slope is calculated as the change in y divided by the change in x, which is (y2 - y1) / (x2 - x1).
Slope between two points

```
type point = float * float

let slope (p1:point) (p2:point) : float =
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    (x1, y1)
```

deconstruct tuple
Slope between two points

```ocaml
type point = float * float

let slope (p1:point) (p2:point) : float =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    ???
```

- **What can we return?**
  - Avoid divide by zero

Diagram: Points `(x1, y1)` and `(x2, y2)` with slope calculation shown.
type point = float * float

let slope (p1:point) (p2:point) : float option =
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    let xd = x2 -. x1 in
    if xd != 0.0 then
        ???
    else
        ???

we need an option type as the result type
Slope between two points

type point = float * float

let slope (p1:point) (p2:point) : float option =
let (x1,y1) = p1 in
let (x2,y2) = p2 in
let xd = x2 -. x1 in
if xd != 0.0 then
  Some ((y2 -. y1) /. xd)
else
  None
type point = float * float

let slope (p1:point) (p2:point) : float option =
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    let xd = x2 -. x1 in
    if xd != 0.0 then
        (y2 -. y1) /. xd
    else
        None
type point = float * float

let slope (p1:point) (p2:point) : float option =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    None

Has type float

WRONG: Type mismatch

Can have type float option
Slope between two points

**Type:**

```plaintext
type point = float * float
```

**Function:**

```plaintext
let slope (p1:point) (p2:point) : float option =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    None
```

**Diagram:**

- Points a, b, and c form a triangle with sides b, c, and d.
- The slope is calculated using the formula 
  \( \frac{y_2 - y_1}{x_2 - x_1} \)
  where 
  \( (x_1, y_1) \) and \( (x_2, y_2) \) are the coordinates of the two points.

**Notes:**

- The calculated result type does not match the declared result type of `float option`.
- The formula for the slope is used correctly, but the calculation of the result type mismatch between `float` and `None` is indicated.

**Example:**

Given points `a = (1.0, 2.0)` and `b = (3.0, 5.0)`, the slope calculation is:

\[
\frac{5.0 - 2.0}{3.0 - 1.0} = \frac{3.0}{2.0}
\]

The result is `1.5`, but since `None` is not a valid floating point number, the result is `None`. The diagram illustrates the triangle and the calculation process.
Remember the typing rule for if

if \( e_1 : \text{bool} \)
and \( e_2 : t \) and \( e_3 : t \) (for some type \( t \))
then if \( e_1 \) then \( e_2 \) else \( e_3 \) : \( t \)

Returning an optional value from an if statement:

if ... then

\[
\text{None} : t \text{ option}
\]

else

\[
\text{Some ( ... )} : t \text{ option}
\]
How do we use an option?

`slope : point -> point -> float option`

returns a float option
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  slope p1 p2

returns a float option; to print we must discover if it is None or Some
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
How do we use an option?

```ocaml
slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  | Some s ->
  | None ->
```

There are two possibilities

Vertical bar separates possibilities
How do we use an option?

```ocaml
slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  | Some s ->
  | None ->
```

The "Some s" pattern includes the variable s

The object between | and -> is called a pattern
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  | Some s ->
  | None ->

You can put a “|” on the first line if you want. It is generally considered better style to do so. When I learned OCaml, that wasn’t an option so I forget to do it a lot...
How do we use an option?

```ocaml
slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
    match slope p1 p2 with
    | Some s ->
        print_string ("Slope: " ^ string_of_float s)
    | None ->
        print_string "Vertical line.\n"
```
Writing Functions Over Typed Data

• Steps to writing functions over typed data:
  1. Write down the function and argument names
  2. Write down argument and result types
  3. Write down some examples (in a comment)
  4. **Deconstruct** input data structures
  5. **Build** new output values
  6. Clean up by identifying repeated patterns

• For option types:

  when the **input** has type `t option`,
  deconstruct with:

  ```
  match ... with
  | None -> ...
  | Some s -> ...
  ```

  when the **output** has type `t option`,
  construct with:

  ```
  Some (...)
  None
  ```
MORE PATTERN MATCHING
Recall the Distance Function

type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
Recall the Distance Function

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

(x2, y2) is an example of a pattern – a pattern for tuples.

So let declarations can contain patterns just like match statements.

The difference is that a match allows you to consider multiple different data shapes.
Recall the Distance Function

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match p1 with
  | (x1,y1) ->
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

There is only 1 possibility when matching a pair
Recall the Distance Function

type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match p1 with
  | (x1,y1) ->
    match p2 with
    | (x2,y2) ->
      sqrt (square (x2 -. x1) +. square (y2 -. y1))

We can nest one match expression inside another. (We can nest any expression inside any other, if the expressions have the right types)
type point = float * float

let distance (p1:point) (p2:point) : float =
   let square x = x *. x in
   match (p1, p2) with
   | ((x1,y1), (x2, y2)) ->
      sqrt (square (x2 -. x1) +. square (y2 -. y1))

Pattern for a pair of pairs:  ((variable, variable), (variable, variable))
All the variable names in the pattern must be different.
BETTER STYLE: COMPLEX PATTERNS

type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    match (p1, p2) with
        | (p3, p4) ->
            let (x1, y1) = p3 in
            let (x2, y2) = p4 in
            sqrt (square (x2 -. x1) +. square (y2 -. y1))

A pattern must be **consistent with** the type of the expression
in between **match ... with**
We use (p3, p4) here instead of ((x1, y1), (x2, y2))
Pattern-matching in function parameters

```plaintext
type point = float * float

let distance ((x1,y1):point) ((x2,y2):point) : float =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

Function parameters are patterns too!
What's the best style?

```ocaml
let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

```ocaml
let distance ((x1,y1):point) ((x2,y2):point) : float =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

Either of these is reasonably clear and compact.
Code with unnecessary nested matches/lets is particularly ugly to read.
You'll be judged on code style in this class.
What’s the best style?

This is how I'd do it ... the types for tuples + the tuple patterns are a little ugly/verbose ... but for now in class, use the explicit type annotations. We will loosen things up later in the semester.

```ocaml
let distance (x1,y1) (x2,y2) =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```
type point = float * float

(* returns a nearby point in the graph if one exists *)
nearby : graph -> point -> point option

let printer (g:graph) (p:point) : unit =
    match nearby g p with
    | None -> print_string "could not find one\n"
    | Some (x,y) ->
      print_float x;
      print_string ", ";
      print_float y;
      print_newline();
Other Patterns

Constant values can be used as patterns

```ocaml
let small_prime (n:int) : bool = 
    match n with
    | 2 -> true
    | 3 -> true
    | 5 -> true
    | _  -> false
```

```ocaml
let iffy (b:bool) : int = 
    match b with
    | true  -> 0
    | false -> 1
```

the underscore pattern matches anything
it is the "don't care" pattern
INDUCTIVE THINKING
An *inductive data type* $T$ is a data type defined by:

- a collection of base cases
  - that don’t refer to $T$
- a collection of inductive cases that build new data of type $T$ from pre-existing data of type $T$
  - the pre-existing data is guaranteed to be *smaller* than the new values

**Programming principle:**

- solve programming problem for base cases
- solve programming problem for inductive cases *by calling the function recursively on smaller data and assuming your function already works correctly on those smaller data values*

**Proving principle:**

- prove program satisfies property $P$ for base cases
- prove inductive cases satisfy property $P$ *by assuming inductive calls on smaller data values satisfy property $P*
LISTS: AN INDUCTIVE DATA TYPE
Lists are Inductive Data

In OCaml, a list value is:

- \([ \ ]\) (the empty list)
- \(v :: vs\) (a value \(v\) followed by a shorter list of values \(vs\))
Lists are Inductive Data

In OCaml, a list value is:

- [ ] (the empty list)
- v :: vs (a value v followed by a shorter list of values vs)

An example:

- 2 :: 3 :: 5 :: [ ] has type int list
- is the same as: 2 :: (3 :: (5 :: [ ]))
- "::" is called "cons"

An alternative syntax ("syntactic sugar" for lists):

- [2; 3; 5]
- But this is just a shorthand for 2 :: 3 :: 5 :: []. If you ever get confused fall back on the 2 basic constructors: :: and []
Typing Lists

Typing rules for lists:

(1) [ ] may have any list type t list

(2) if e1 : t and e2 : t list then (e1 :: e2) : t list
Typing Lists

Typing rules for lists:

(1) \[ \] may have any list type \( t \text{ list} \)

(2) if \( e_1 : t \) and \( e_2 : t \text{ list} \)
then \((e_1 :: e_2) : t \text{ list}\)

More examples:

(1 + 2) :: (3 + 4) :: [ ] : ??

(2 :: [ ]) :: (5 :: 6 :: [ ]) :: [ ] : ??

Typing Lists

Typing rules for lists:

(1) \[ \text{[]} \] may have any list type \( t \text{ list} \)

(2) \text{if } e1 : t \text{ and } e2 : t \text{ list} \text{ then } (e1 :: e2) : t \text{ list}

More examples:

(1 + 2) :: (3 + 4) :: [ ] : int list

(2 :: [ ]) :: (5 :: 6 :: [ ]) :: [ ] : int list list

[ [2]; [5; 6] ] : int list list

(Remember that the 3\text{rd} example is an abbreviation for the 2\text{nd})
Another Example

What type does this have?

Another Example

What type does this have?

```
# [2] :: [3];;
Error: This expression has type int but an expression was expected of type int list
#
```
Another Example

What type does this have?


int list

int list

Give me a simple fix that makes the expression type check?
Another Example

What type does this have?


int list

int list

Give me a simple fix that makes the expression type check?

Either: 2 :: [ 3 ] : int list

Analyzing Lists

Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

(* return Some v, if v is the first list element; return None, if the list is empty *)

let head (xs : int list) : int option =
Analyzing Lists

Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

(* return Some v, if v is the first list element; return None, if the list is empty *)

let head (xs : int list) : int option =
  match xs with
  | [] ->
  | hd :: _ ->

we don't care about the contents of the tail of the list so we use the underscore
Analyzing Lists

Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

(* return Some v, if v is the first list element; return None, if the list is empty *)

let head (xs : int list) : int option =
  match xs with
  | [] -> None
  | hd :: _ -> Some hd

This function isn't recursive -- we only extracted a small, fixed amount of information from the list -- the first element
A more interesting example

(* Given a list of pairs of integers, produce the list of products of the pairs
prods [(2,3); (4,7); (5,2)] == [6; 28; 10] *)
A more interesting example

(* Given a list of pairs of integers, produce the list of products of the pairs

prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
(* Given a list of pairs of integers, produce the list of products of the pairs

prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
match xs with
| [] ->
| (x,y) :: tl ->
A more interesting example

(* Given a list of pairs of integers, produce the list of products of the pairs

prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
match xs with
| [] -> []
| (x,y) :: tl ->
A more interesting example

(* Given a list of pairs of integers, produce the list of products of the pairs

prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> ?? :: ??

the result type is int list, so we can speculate that we should create a list
A more interesting example

(* Given a list of pairs of integers, produce the list of products of the pairs

prods [(2,3); (4,7); (5,2)] == [6; 28; 10] *)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: ??

the first element is the product
(* Given a list of pairs of integers, produce the list of products of the pairs

prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: ??

to complete the job, we must compute the products for the rest of the list
A more interesting example

(* Given a list of pairs of integers,
produce the list of products of the pairs

\[
\text{prods } [(2,3); (4,7); (5,2)] == [6; 28; 10]
*
)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: prods tl
Three Parts to Constructing a Function

(1) Think about how to \textit{break down} the input into cases:

$$
\text{let rec prods (xs : (int*int) list) : int list =}
\begin{align*}
\text{match xs with} & \\
| [] & \to \ldots & \\
| (x,y) :: tl & \to \ldots \text{ prods tl} \ldots
\end{align*}
$$

This assumption is called the \textit{Induction Hypothesis}. You’ll use it to prove your program correct.

(2) \textit{Assume} the recursive call on smaller data is correct.

(3) Use the result of the recursive call to \textit{build} correct answer.

$$
\text{let rec prods (xs : (int*int) list) : int list =}
\begin{align*}
\text{...} & \\
| (x,y) :: tl & \to \ldots \text{ prods tl} \ldots
\end{align*}
$$
Another example: zip

(* Given two lists of integers, return None if the lists are different lengths otherwise stitch the lists together to create Some of a list of pairs)

zip [2; 3] [4; 5] == Some [(2,4); (3,5)]
zip [5; 3] [4] == None
zip [4; 5; 6] [8; 9; 10; 11; 12] == None

(Give it a try.)
Another example: zip

```ocaml
let rec zip (xs : int list) (ys : int list) :
    (int * int) list option =
```
let rec zip (xs : int list) (ys : int list) : (int * int) list option =

match (xs, ys) with
Another example: zip

let rec zip (xs : int list) (ys : int list) : (int * int) list option =

  match (xs, ys) with
  | ([], []) ->
  | ([], y::ys') ->
  | (x::xs', []) ->
  | (x::xs', y::ys') ->
Another example: zip

```ocaml
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') ->
  | (x::xs', []) ->
  | (x::xs', y::ys') ->
```

let rec zip (xs : int list) (ys : int list) : (int * int) list option =

match (xs, ys) with
| ([], []) -> Some []
| ([], y::ys') -> None
| (x::xs', []) -> None
| (x::xs', y::ys') ->
let rec zip (xs : int list) (ys : int list) : (int * int) list option =

match (xs, ys) with
| ([], []) -> Some []
| ([], y::ys') -> None
| (x::xs', []) -> None
| (x::xs', y::ys') -> (x, y) :: zip xs' ys'

is this ok?
let rec zip (xs : int list) (ys : int list) : (int * int) list option =

match (xs, ys) with
| ([], []) -> Some []
| ([], y::ys') -> None
| (x::xs', []) -> None
| (x::xs', y::ys') -> (x, y) :: zip xs' ys'

No! zip returns a list option, not a list!
We need to match it and decide if it is Some or None.
let rec zip (xs : int list) (ys : int list) : (int * int) list option =

match (xs, ys) with
| ([], []) -> Some []
| ([], y::ys') -> None
| (x::xs', []) -> None
| (x::xs', y::ys') ->
  (match zip xs' ys' with
   None -> None
   | Some zs -> (x,y) :: zs)
let rec zip (xs : int list) (ys : int list) : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') ->
    (match zip xs' ys' with
     None -> None
     | Some zs -> Some ((x,y) :: zs))
let rec zip (xs : int list) (ys : int list) : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | (x::xs', y::ys') ->
    (match zip xs' ys' with
      None -> None
      | Some zs -> Some ((x,y) :: zs))
  | (_, _) -> None

Clean up.
Reorganize the cases.
Pattern matching proceeds in order.
let rec sum (xs : int list) : int =
    match xs with
    | hd::tl -> hd + sum tl
A bad list example

let rec sum (xs : int list) : int =
    match xs with
    | hd::tl -> hd + sum tl

# Characters 39-78:
..match xs with
    hd :: tl -> hd + sum tl..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: []
val sum : int list -> int = <fun>
INSERTION SORT
Recall Insertion Sort

At any point during the insertion sort:

- some initial segment of the array will be sorted
- the rest of the array will be in the same (unsorted) order as it was originally

![Array Illustration]
Recall Insertion Sort

At any point during the insertion sort:

- some initial segment of the array will be sorted
- the rest of the array will be in the same (unsorted) order as it was originally

At each step, take the next item in the array and insert it in order into the sorted portion of the list
Insertion Sort With Lists

The algorithm is similar, except instead of *one array*, we will maintain *two lists*, a sorted list and an unsorted list.

We'll factor the algorithm:

- a function to insert into a sorted list
- a sorting function that repeatedly inserts
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
let rec insert (x : int) (xs : int list) : int list =
  match xs with
  | []   ->
  | hd :: tl ->

a familiar pattern: analyze the list by cases
(* insert x into sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
match xs with
| [] -> [x]
| hd :: tl ->
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
  match xs with
  | [] -> [x]
  | hd :: tl ->
    if hd < x then
      hd :: insert x tl
    build a new list with:
    • hd at the beginning
    • the result of inserting x in to the tail of the list afterwards
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
  match xs with
  | [] -> [x]
  | hd :: tl ->
    if hd < x then
      hd :: insert x tl
    else
      x :: xs

put x on the front of the list, the rest of the list follows
Insertion Sort

```plaintext
type il = int list

insert : int -> il -> il

(* insertion sort *)

let rec insert_sort(xs : il) : il =
```

type il = int list

insert : int -> il -> il

(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =

    in
type il = int list

insert : int -> il -> il

(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =

    in
    aux [] xs
type il = int list

insert : int -> il -> il

(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =
  match unsorted with
  | [] ->
  | hd :: tl ->
  in
  aux [] xs
type il = int list

insert : int -> il -> il

(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =
    match unsorted with
    | [] -> sorted
    | hd :: tl -> aux (insert hd sorted) tl
  in
  aux [] xs
A SHORT JAVA RANT
public class Pair {
    public int x;
    public int y;

    public Pair (int a, int b) {
        x = a;
        y = b;
    }
}

public class User {
    public Pair swap (Pair p1) {
        Pair p2 =
            new Pair(p1.y, p1.x);
        return p2;
    }
}

What could go wrong?
A Paucity of Types

The input \texttt{p1} to swap may be \texttt{null} and we forgot to check.

Java has no way to define a pair data structure that is \textit{just a pair}.

\textit{How many students in the class have seen an accidental null pointer exception thrown in their Java code?}
In OCaml, if a pair may be null it is a pair option:

```ocaml
type java_pair = (int * int) option
```
From Java Pairs to OCaml Pairs

In OCaml, if a pair may be null it is a pair option:

```ocaml
type java_pair = (int * int) option
```

And if you write code like this:

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
  let (x,y) = p in
  (y,x)
```
From Java Pairs to OCaml Pairs

In OCaml, if a pair may be null it is a pair option:

```ocaml
type java_pair = (int * int) option
```

And if you write code like this:

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
  let (x,y) = p in
  (y,x)
```

You get a helpful error message like this:

```ocaml
# ... Characters 91-92:
  let (x,y) = p in (y,x);;
  ^
Error: This expression has type java_pair = (int * int) option
  but an expression was expected of type 'a * 'b
```
From Java Pairs to OCaml Pairs

```ocaml
type java_pair = (int * int) option
```

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
  match p with
  | Some (x,y) -> Some (y,x)
```
From Java Pairs to OCaml Pairs

```ocaml
type java_pair = (int * int) option

let swap_java_pair (p:java_pair) : java_pair =
  match p with
  | Some (x,y) -> Some (y,x)
```

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
  match p with
  | Some (x,y) -> Some (y,x)
```

**Warning 8:** this pattern-matching is not exhaustive. Here is an example of a value that is not matched: `None`
From Java Pairs to OCaml Pairs

```ocaml
type java_pair = (int * int) option

let swap_java_pair (p:java_pair) : java_pair =
match p with
| Some (x,y) -> Some (y,x)
```

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
match p with
| None -> None
| Some (x,y) -> Some (y,x)
```

An easy fix!
Moreover, your pairs are probably almost never null!

Defensive programming & always checking for null is AnNOyinG
There just isn't always some "good thing" for a function to do when it receives a bad input, like a null pointer.

In OCaml, all these issues disappear when you use the proper type for a pair and that type contains no "extra junk".

```ocaml
type pair = int * int

let swap (p:pair) : pair =
  let (x,y) = p in (y,x)
```

Once you know OCaml, it is hard to write swap incorrectly. Your bullet-proof code is much simpler than in Java.
Java has a paucity of types

– There is no type to describe just the pairs
– There is no type to describe just the triples
– There is no type to describe the pairs of pairs
– There is no type ...

OCaml has many more types

– use option when things may be null
– do not use option when things are not null
– OCaml types describe data structures more precisely
  • programmers have fewer cases to worry about
  • entire classes of errors just go away
  • type checking and pattern analysis help prevent programmers from ever forgetting about a case
Summary of Java Pair Rant

Java has a paucity of types
- There is no type to describe just the pairs
- There is no type to describe just the triples
- There is no type to describe the pairs of pairs
- There is no type ...

OCaml has many more types
- Use option when things may be null
- Do not use option when things are not null
- OCaml types describe data structures more precisely
  - Programmers have fewer cases to worry about
  - Entire classes of errors just go away
  - Type checking and pattern analysis help prevent programmers from ever forgetting about a case

SCORE: OCAML 1, JAVA 0
Example problems to practice

• Write a function to sum the elements of a list
  – sum [1; 2; 3] ==> 6

• Write a function to append two lists
  – append [1;2;3] [4;5;6] ==> [1;2;3;4;5;6]

• Write a function to reverse a list
  – rev [1;2;3] ==> [3;2;1]

• Write a function to turn a list of pairs into a pair of lists
  – split [(1,2); (3,4); (5,6)] ==> ([1;3;5], [2;4;6])

• Write a function that returns all prefixes of a list
  – prefixes [1;2;3] ==> [[]; [1]; [1;2]; [1;2;3]]

• suffixes...