Administration

• We’ll announce on Piazza when you can start an assignment
  – don’t start early as there may be changes!
  – sign up for Piazza!
  – Assignment 1 due at 11:59 tonight!

• Program style guide:

• Read notes:
  – functional basics, type-checking, typed programming
  – thinking recursively (today)
  – Real World OCaml Chapter 2, 3
Typed Functional Programming

• Functional programs operate by:
  – extracting information from their arguments and then
  – producing new values

• So far, we've defined nonrecursive functions in this style to analyze pairs and optional values

• Why? Because recursive functions typically operate on recursive data
  – Pairs are not recursive -- we need only do a small, (statically) predictable amount of work to get at the information these structures contain
  – Lists and natural numbers can be viewed as recursive
    • not surprisingly, you’ve defined recursive functions over numbers!
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 values of type \( T \) from pre-existing data of type \( T \)

**Programming principle:**

- solve programming problem for base cases
- solve programming problem for inductive cases by calling function recursively (inductively) on *smaller* data value

**Proving principle:**

- prove program satisfies property \( P \) for base cases
- prove inductive case satisfies property \( P \) assuming inductive call on *smaller* data value satisfies property \( P \)
LISTS: AN INDUCTIVE DATA TYPE
Lists are Recursive Data

• In O'Caml, 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 O'Caml, 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 \( t \text{ list} \)

  \[ [ ] : T \text{ list} \]

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

  \[
  \begin{align*}
  e_1 : T & \quad e_2 : T \text{ list} \\
  e_1 :: e_2 & : T \text{ list}
  \end{align*}
  \]
Typing Lists

• Typing rules for lists:

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

  \[
  (1) \quad [\ ] : T\ list
  \]

  (2) if \(e_1 : t\) and \(e_2 : t\ List\)
  then \(e_1 :: e_2 : t\ list\)

  \[
  (2) \quad \text{if } e_1 : T \text{ and } e_2 : T\ list \text{ then } e_1 :: e_2 : T\ list
  \]

• More examples:

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

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

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

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

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) \((1 + 2) :: (3 + 4) :: [\ ]\) : \text{int list}

  (2) \((2 :: [\ ]) :: (5 :: 6 :: [\ ]) :: [\ ]\) : \text{int list list}

  \([ [2]; [5; 6] ]\) : \text{int list list}

  (Remember that the 3\textsuperscript{rd} example is an abbreviation for the 2\textsuperscript{nd})
• What type does this have?

Another Example

• What type does this have?

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

```
```

```
e1:T e2:T list
```

```
e1::e2 : T list
```
Another Example

- What type does this have?

\[
\begin{array}{c}
\text{int list} & \text{int list}
\end{array}
\]

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

\[
e1:T \quad e2:T \text{ list} \\
e1::e2 : T \text{ list}
\]
Another Example

- What type does this have?

\[
\begin{array}{c}
\text{int list} & \text{int list}
\end{array}
\]

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

Either: \[2 :: [3] : \text{int list}\]

Or: \[[2] :: [[3]] : \text{int list list}\]

\_e1:T \_e2:T \text{list}
\_e1::e2 : T \text{list}
Analyzing Lists

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

```ocaml
(* 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

\[
\begin{align*}
\text{let head (xs : int list) : int option =} \\
\text{match xs with} \\
\mid [\] & -> \\
\mid \text{hd :: _} & -> \\
\end{align*}
\]

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

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 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] *)
(* 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 = ;;
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 ->
;;
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
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) :: ??
;;

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

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
;;
Two Parts to Constructing a Function

Think about how to break down the input into cases:

\[
\text{let rec prods (xs : (int*int) list) : int list =}
\]

\[
\begin{align*}
| [] & \rightarrow \ldots \\
| (x,y) :: tl & \rightarrow \ldots \text{ prods tl} \\
\end{align*}
\]

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

Assume the recursive call is correct (i.e.: its result satisfies the property you want).

Use its result to build correct answer.

\[
\begin{align*}
\text{let rec prods (xs : (int*int) list) : int list =} \\
\quad \ldots \\
| (x,y) :: tl & \rightarrow \ldots \text{ prods tl} \ldots
\end{align*}
\]
Recap

Broad steps:
- *break down the input* based on its type into a set of cases
  - there can be more than one way to do this
- *make the assumption* (the *induction hypothesis*) that your recursive function works correctly when called on a *smaller list*
  - you might have to make 0, 1, 2 or more recursive calls
- *build the output* (guided by its type) from the results of recursive calls

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

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

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

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

match (xs, ys) with

;;
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') ->

;;
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?
Another example: zip

```ocaml
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.
Another example: zip

```
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
        )
;;
```

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') ->
    (match zip xs' ys' with
     None -> None
     | Some zs -> Some ((x,y) :: zs)
    );;
Another example: zip

```ocaml
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

-5 -2 3 -4 10 6 7

sorted unsorted
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

-5  -2  3  -4  10  6  7

-5  -4  -2  3  10  6  7

• 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.

- list 1:
  -5 -2 3
  sorted
- list 2:
  -4 10 6 7
  unsorted

• 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 =
(* insert x in to sorted list xs *)

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 in to 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
    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**

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

```ocaml
type il = int list
insert : int -> il -> il

(* insertion sort *)

let rec insert_sort(xs : il) : il = ;;
```
Insertion Sort

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 COUPLE MORE THOUGHTS ON LISTS
The (Single) List Programming Paradigm

• Recall that a list is either:
  – [ ] (the empty list)
  – v :: vs (a value v followed by a previously constructed list vs)

• Some examples:

```ml
let l0 = [];; (* length is 0 *)
let l1 = 1::l0;; (* length is 1 *)
let l2 = 2::l1;; (* length is 2 *)
let l3 = 3::l2;; (* length is 3 *)
...
```
Consider the following picture. How long is the linked structure? Can we build a value with type `int list` to represent it?
Consider This Picture

• How long is it? **Infinitely long?**
• Can we build a value with type `int list` to represent it? **No!**
  – all values with type `int list` have finite length
• Is it a good thing that the type list does not contain any infinitely long lists? Yes!

• A terminating list-processing scheme:

```ocaml
define f (xs : int list) : int =
  match xs with
  [ ] -> … do something not recursive …
  | hd::tail -> … f tail …
;;
```

terminates because f only called recursively on smaller lists
A Loopy Program

let rec loop (xs : int list) : int =
  match xs with
  | [] -> 0
  | hd::tail -> hd + loop (0::tail)
;;

Does this program terminate?
A Loopy Program

Does this program terminate? No! Why not? We call loop recursively on (0::tail). This list is the same size as the original list -- not smaller.

```ocaml
let loop (xs : int list) : int =
  match xs with
  [|] -> []
  | hd::tail -> hd + loop (0::tail)
;;
```
ML has a \textit{strong type system}

- ML \textit{types say a lot} about the set of values that inhabit them

In this case, the tail of the list is \textit{always} shorter than the whole list

This makes it easy to write functions that terminate; \textit{it would be harder if you had to consider more cases}, such as the case that the tail of a list might loop back on itself. \textit{Moreover OCaml hits you over the head to tell you what the only 2 cases are!}

Note: Just because the list type excludes cyclic structures does not mean that an ML program can't build a cyclic data structure if it wants to. (We'll do that later in the course.)
Rant #2: Imperative lists

- One week from today, ask yourself: Which is easier:
  - Programming with immutable lists in ML?
  - Programming with pointers and mutable cells in C/Java

SCORE: OCAML 2, JAVA 0
I want to build a perfect HO-scale (1/87) model train layout of my town.

Why not?

Because it'd include a little 1" replica of your house.

So? That'd be cool. I'd make tiny replicas of my rooms, my furniture—

—and your train layout?

---

The Matryoshka limit. It is impossible to nest more than six HO layouts.

My god.

Yeah, it's the second rule of model train layouts: no nesting.

...What's the first rule?

"Do not talk about model train layouts." That rule was actually voted in by our friends and families. Philistines.
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...

Is that a tree or a DAG? Does it matter?
ANOTHER INDUCTIVE DATA TYPE: THE NATURAL NUMBERS
Natural Numbers

• Natural numbers are a lot like lists
  – both can be defined inductively

• A natural number \( n \) is either
  – 0, or
  – \( m + 1 \) where \( m \) is a smaller natural number

• Functions over naturals \( n \) must consider both cases
  – programming the base case 0 is usually easy
  – programming the inductive case \( (m+1) \) will often involve recursive calls over smaller numbers

• OCaml doesn't have a built-in type "nat" so we will use "int" instead for now ...
  – “int” has too many values in it (and also not enough)
  – later in the course we could define an \emph{abstract type} that contains exactly the natural numbers
An Example

(* precondition: n is a natural number
return double the input *)

let rec double_nat (n : int) : int =

;;

By definition of naturals:
• n = 0 or
• n = m+1 for some nat m
An Example

(* precondition: n is a natural number
  return double the input *)

let rec double_nat (n : int) : int =
  match n with
  | 0   ->
  | _   ->
  ;;

By definition of naturals:
  • n = 0 or
  • n = m+1 for some nat m
An Example

(* precondition: n is a natural number return double the input *)

let rec double_nat (n : int) : int =
  match n with
  | 0 -> 0
  | _ ->
  ;;

solve easy base case first
consider:
what number is double 0?

By definition of naturals:
• n = 0 or
• n = m+1 for some nat m
An Example

(* precondition: n is a natural number
return double the input *)

let rec double_nat (n : int) : int =
    match n with
    | 0 -> 0
    | _ -> ????
    ;;

assume double_nat m is correct where n = m+1
that’s the inductive hypothesis

By definition of naturals:
• \( n = 0 \) or
• \( n = m+1 \) for some nat \( m \)
An Example

(* precondition: n is a natural number
return double the input *)

let rec double_nat (n : int) : int =
  match n with
  | 0 -> 0
  | _ -> 2 + double_nat (n-1)
;;

By definition of naturals:
• n = 0 or
• n = m+1 for some nat m

assume double_nat m is correct
where n = m+1

that's the inductive hypothesis

I wish I had a pattern (m+1) ... but OCaml doesn’t have it. So I use n-1 to get m.
let double (n : int) : int =

let rec double_nat (n : int) : int =
  match n with
  0 -> 0
  | n -> 2 + double_nat (n-1)
in

if n < 0 then
  failwith "negative input!"
else
double_nat n
;;
More than one way to decompose naturals

A natural n is either:
- 0,
- \( m+1 \), where m is a natural

unary decomposition

A natural n is either:
- 0,
- 1,
- \( m+2 \), where m is a natural

unary even/odd decomposition

A natural n is either:
- 0,
- \( m*2 \)
- \( m*2+1 \)

binary decomposition
(there’s a little problem here with a redundant representation; what is it?)
More than one way to decompose lists

A list \(xs\) is either:
- \([\,]\),
- \(x::xs\), where \(ys\) is a list

\[\text{unary decomposition}\]

A list \(xs\) is either:
- \([\,]\),
- \([x]\),
- \(x::y::ys\), where \(ys\) is a list

\[\text{unary even/odd decomposition}\]

A list \(xs\) is either:
- \([\,]\),
- \(a@b\)
- \(x :: (a@b)\)

\[\text{where } a \text{ and } b \text{ are lists of the same length; recall that } @ \text{ is list-concat}\]
Summary

• Instead of while or for loops, functional programmers use recursive functions

• These functions operate by:
  – decomposing the input data
  – considering all cases
  – some cases are base cases, which do not require recursive calls
  – some cases are inductive cases, which require recursive calls on smaller arguments

• We've seen:
  – lists with cases:
    • (1) empty list, (2) a list with one or more elements
  – natural numbers with cases:
    • (1) zero    (2) m+1
  – we'll see many more examples throughout the course