INDUCTIVE THINKING
Inductive Programming and Proving

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 \( \text{int list} \)
- 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 \textit{constructors}, :: and []
Typing Rules for Lists:

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

2. if \(e_1 : t\) and \(e_2 : t\ list\)
then \(e_1 :: e_2 : t\ list\)
Typing Lists

• Typing rules for lists:

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

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

• More examples:

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

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

\([ [2]; [5; 6] ]\) : ??
Typing Lists

• Typing rules for lists:

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

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

• More examples:

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

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

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

(Remember that the 3rd example is an abbreviation for the 2nd)
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
```

```
rule: e1 :: e2 : t list if e1 : t and e2 : t list
```

```bash
# [2] :: [3];;
```
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?

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

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

Either: \[ 2 :: [ 3 ] \] : 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

```ml
(* 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

```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 =
  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]
*)
(* 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 ->
    ;;
(* 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

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

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

reasoning process:
• assume prods computes correctly on the smaller list tl
• conclude therefore that \((x * y) :: \text{prods } tl\) is correct for the entire 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
;;

- Next: test it. What inputs should we test it on?
Note the strategy

• 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

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

;;
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') ->
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') -> (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.
Another example: zip

```ml
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)
```

Closer, but no cigar.
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 -> 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
;;
let rec sum (xs : int list) : int =
  match xs with
  | hd::tl -> hd + sum tl
;;

# Characters 39-78:
..match xs with
  x :: xs -> x + sum xs..
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
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 [-2 3 -4 10 6 7]

sorted unsorted

• At each step, take the next item in the array and insert it in order into the sorted portion of the list

```
-5  -4  -2  3  10  6  7
```

-5 [-4 -2 3 10 6 7]

sorted unsorted
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

```
list 1:
-5  -2  3

sorted

list 2:
-4  10  6  7

unsorted
```
(* insert x in to sorted list xs *)

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 = ;;
(* insert x into 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 the empty list
(* 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
      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
   ;;
type il = int list

insert : int -> il -> il

(* insertion sort *)

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

```ocaml
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:

```plaintext
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
The List Type

- Is it a good thing that the type list does not contain any infinitely long lists? Yes!

- A terminating list-processing scheme:

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

Does this program terminate?
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)  ;;
```
Take-home Message

ML has a *strong type system*

• ML *types say a lot* about the set of values that inhabit them

In this case, the tail of the list is *always* shorter than the whole list

This makes it easy to write functions that terminate; 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

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.)
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 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]]
PROGRAMMING WITH NATURAL NUMBERS
Natural Numbers

• Natural numbers are a lot like lists
  – both can be defined recursively (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 ...
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)

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 \( m \)

I wish I had a pattern \((m+1)\) ... but OCaml doesn’t have it. So I use \( n-1 \) to get \( m \).
(* fail if the input is negative  
   double the input if it is non-negative *)

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

An Example

nest `double_nat` so it can only be called by `double`

raises exception

protect precondition of `double_nat` by wrapping it with dynamic check

later we will see how to create a static guarantee using types
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:
- $0$,
- $m\times2$
- $m\times2+1$

binary decomposition
More than one way to decompose lists

A list \( xs \) is either:
- \( \text{} \)
- \( x::xs \), where \( ys \) is a list

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

A natural \( n \) is either:
- \( 0 \)
- \( m*2 \)
- \( m*2+1 \)
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 \textit{base cases}, which do not require recursive calls
- some cases are \textit{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
A SHORT JAVA RANT
Definition and Use of Java Pairs

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?
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;
    }
}

The input p1 to swap may be null and we forgot to check.
Java has no way to define a pair data structure that is just a pair.
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
```
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)
```
In OCaml, if a pair may be null it is a pair option:

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

And if you write code like this:

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

You get a *helpful* error message like this:

```bash
# ... 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

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 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

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

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

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

**OCaml to the rescue!**

```
..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 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!

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
    match p with
    | None -> None
    | Some (x,y) -> Some (y,x)
```
Moreover, your pairs are probably almost never null!

Defensive programming & always checking for null is AnNOyinG
From Java Pairs to OCaml Pairs

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
Java has a paucity of types

- There is no type to describe just the pairs
- There is no type to describe just the pairs of pairs
- There is no type to describe just the pairs of pairs of pairs

OCaml types describe data structures more precisely

SCORE: OCAML 1, JAVA 0
Java has a paucity of types
  – but at least when you forget something, it throws an exception instead of silently going off the trolley!

If you forget to check for null pointer in a C program,
  – no type-check error at compile time
  – no exception at run time
  – it might crash right away (that would be best), or
  – it might permit a buffer-overrun (or similar) vulnerability
  – so the hackers pwn you!
Java has a paucity of types

- but at least when you forget something, it throws an exception instead of silently going off the trolley!

If you forget to check for null pointer in a C program,
- no type-check error at compile time,
- no exception at run time,
- it might crash right away (that would be best), or
- it might permit a buffer-overrun (or similar) vulnerability so the hackers pwn you!

Summary of C, C++ rant
Reading Assignments

• Lecture Notes 04: Thinking Inductively

• Optional: Book “Real World OCaml”
  • Chapter 3