CS 320 : Functional Programming in Ocaml

(based on slides from David Walker, Princeton, Lukasz Ziarek, Buffalo and myself.)

Marco Gaboardi

MSC 116

gaboardi@bu.edu

Announcements

- You should be able to use GradeScope now. If you have some difficulty with submitting your homework to GradeScope, send a message to Piazza in which you describe the difficulty.
- First programming assignment is due ~~Friday, February 7,~~ Tuesday, February 11, no later than 11:59 pm.

In the previous classes...

Type Soundness

"Well typed programs do not go wrong"

Programming languages with this property have *sound* type systems. They are called *safe* languages.

Safe languages are generally *immune* to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc.

(but not immune to all bugs!)

Safe languages: ML, Java, Python, …

Unsafe languages: C, C++, Pascal

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```


-->

```
let y = 4     in
y * 2
```

substitute
4 for y

-->

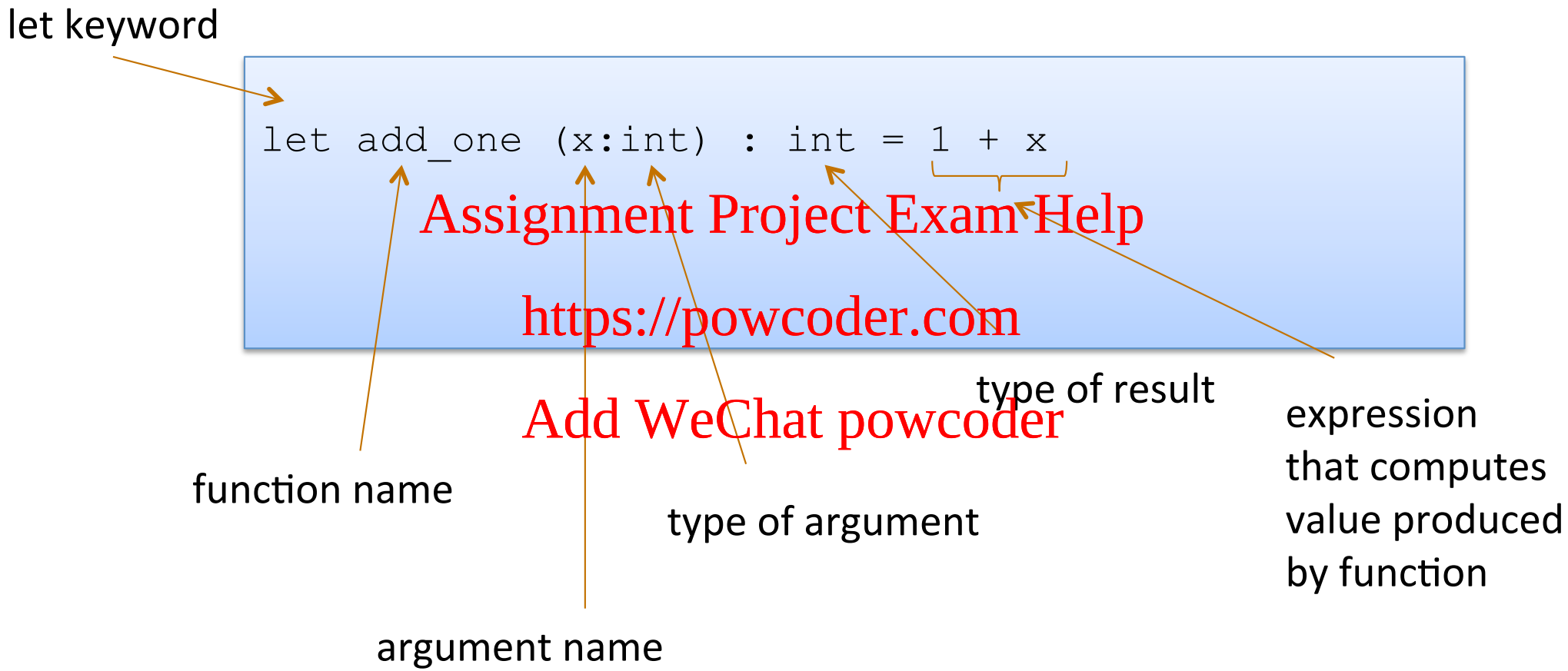
```
4 * 2
```

-->

```
8
```

Moral: Let operates by *substituting* computed values for variables

Defining functions



Note: recursive functions with begin with "**let rec**"

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) : int -> int
```

Tuples

- To use a tuple, we extract its components
- General case:


```
let (id1, id2, ..., idn) = e1 in e2
```


- An example:

```
let (x,y) = (2,4) in x + x + y
```

Unit

- Unit is the tuple with zero fields!

  - the unit value is written with an pair of parens
  - there are no other values with this type!

- Why is the unit type and value useful?
- Every expression has a type:

```
(print_string "hello world\n")  :    unit
```

- Expressions executed for their *effect* return the unit value

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
```

Input and Output Channels

The normal way of opening a file in OCaml returns a **channel**. There are two kinds of channels:

- channels that write to a file: type `out_channel`
- channels that read from a file: type `in_channel`

Four operations that will be useful are:

- Open input file: `open_in: string -> in_channel`
- Open out file: `open_out: string -> out_channel`
- Close input file: `close_in: in_channel -> unit`
- Close input file: `close_out: out_channel -> unit`

If you want to use a channel, you can use `let`, as usual.

Lists are Recursive Data

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

Learning Goals for today

- More on Inductive data types
  Lists
- Functional programming
  Writing more complex functions
  Factoring your code

More Inductive Thinking

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 list

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

- More examples:

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

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

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

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

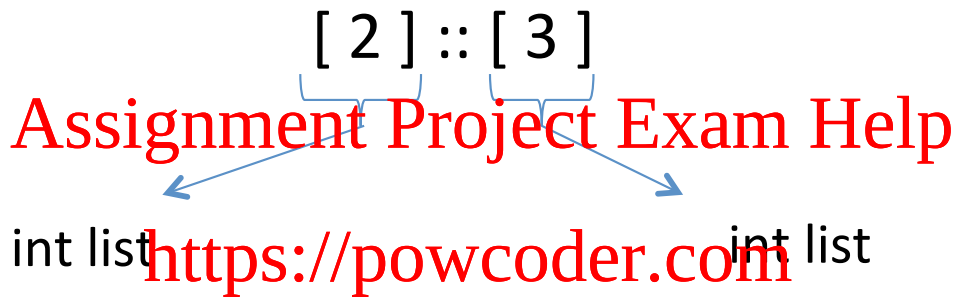
Another Example

- What type does this have?

[2] :: [3]

Another Example

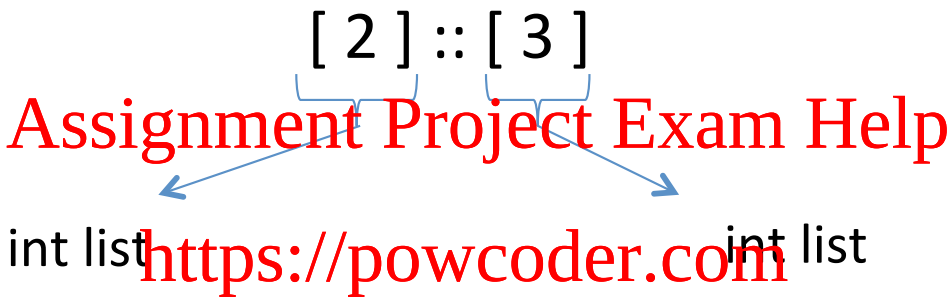
- What type does this have?




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

Another Example

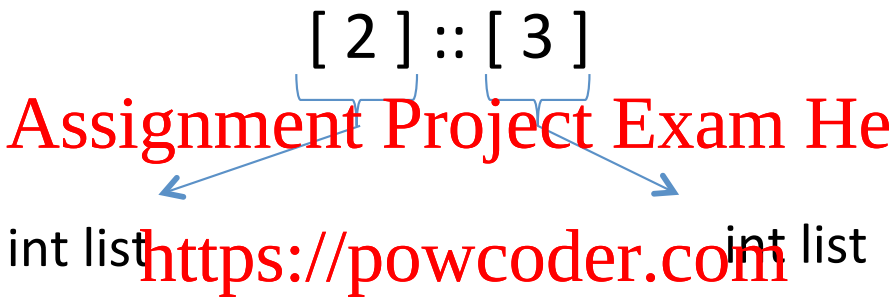
- What type does this have?



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

Another Example

- What type does this have?



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

Either: 2 :: [3] : int list

Or: [2] :: [[3]] : int list list

TopHat Q1-Q7

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 =
```

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
```

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
```

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

```
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?

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'
```


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?

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))
```

Another example: zip

```
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.

A bad list example

```
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
```



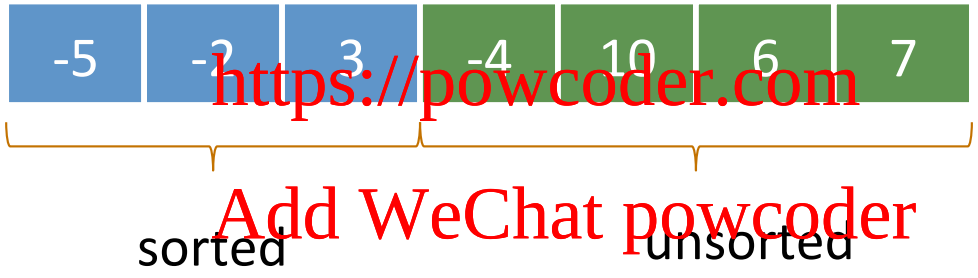
```
#       Characters 39-78:
  ..match xs with
      hd :: tl -> hd + sum tl..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: []
val sum : int list -> int = <fun>
```

TopHat Q8-Q9

An example: insertion sort

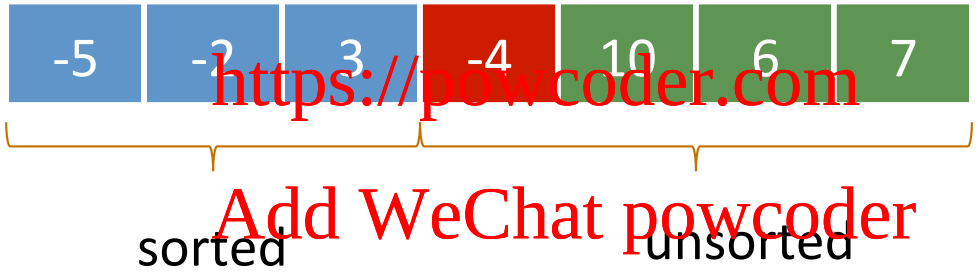
Recall Insertion Sort

- 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

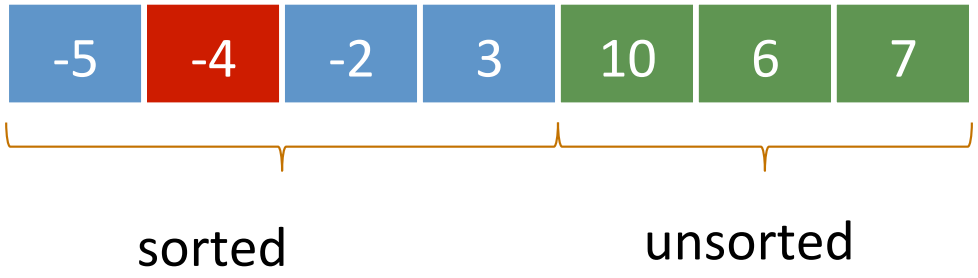


Recall Insertion Sort

- 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

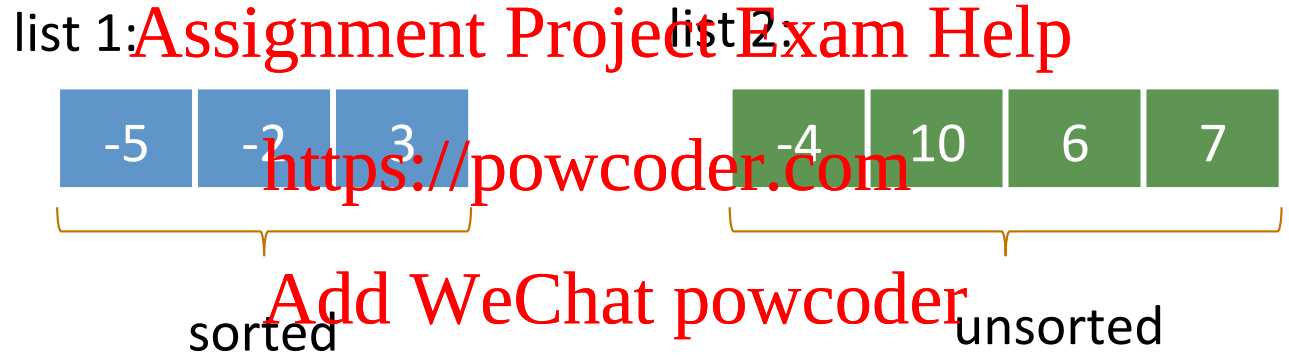


- At each step, take the next item in the array and insert it in order into the sorted portion of the list



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

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

let rec insert (x : int) (xs : int list) : int list =
```

Insert

```
(* 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

```
(* 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 into the empty list

Insert

```
(* 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

```
(* 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

```
type il = int list

insert : int -> il -> il


(* insertion sort *)

let rec insert_sort xs : int list =
```

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
```

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
  aux [] xs
```

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 =
    match unsorted with
    | [] ->
    | hd :: tl ->
  in
  aux [] xs
```

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 =
    match unsorted with
    | [] -> sorted
    | hd :: tl -> aux (insert hd sorted) tl
  in
  aux [] xs
```

More functional programming

Some Design & Coding Rules

- *Laziness* can be a really good force in design.
- Never write the same code twice.
  - factor out the common bits into a reusable procedure.
  - better, use someone else's (well-tested, well-documented, and well-maintained) procedure.
- Why is this a good idea?
  - why don't we just cut-and-paste snippets of code using the editor instead of creating new functions?
  - find and fix a bug in one copy, have to fix in all of them.
  - decide to change the functionality, have to track down all of the places where it gets used.

Factoring Code in OCaml

Consider these definitions:

```
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)
```



```
let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```

Factoring Code in OCaml

Consider these definitions:

```
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)
```



```
let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```


The code is almost identical – factor it out!

Factoring Code in OCaml

A *higher-order* function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```

Factoring Code in OCaml

A *higher-order* function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```


Uses of the function:

```
let inc x = x+1
let inc_all xs = map inc xs
```

Factoring Code in OCaml

A *higher-order* function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```


Uses of the function:

```
let inc x = x+1
let inc_all xs = map inc xs

let square y = y*y
let square_all xs = map square xs
```

Writing little functions like inc just so we call map is a pain.

Factoring Code in OCaml

A higher-order function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```


We can use an *anonymous* function instead.

Originally, Church wrote this function using λ instead of **fun**: (λx. x+1) or (λx. x*x)

Uses of the function:

```
let inc_all xs = map (fun x -> x + 1) xs

let square_all xs = map (fun y -> y * y) xs
```

TopHat Q10-Q13

Summary

- More on Inductive data types
  Lists
- Functional programming
  Writing more complex functions
  Factoring your code