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

- We are setting up GradeScope this week … and before the deadline of the first programming assignment.

- First programming assignment is due Friday, February 7, 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

-->

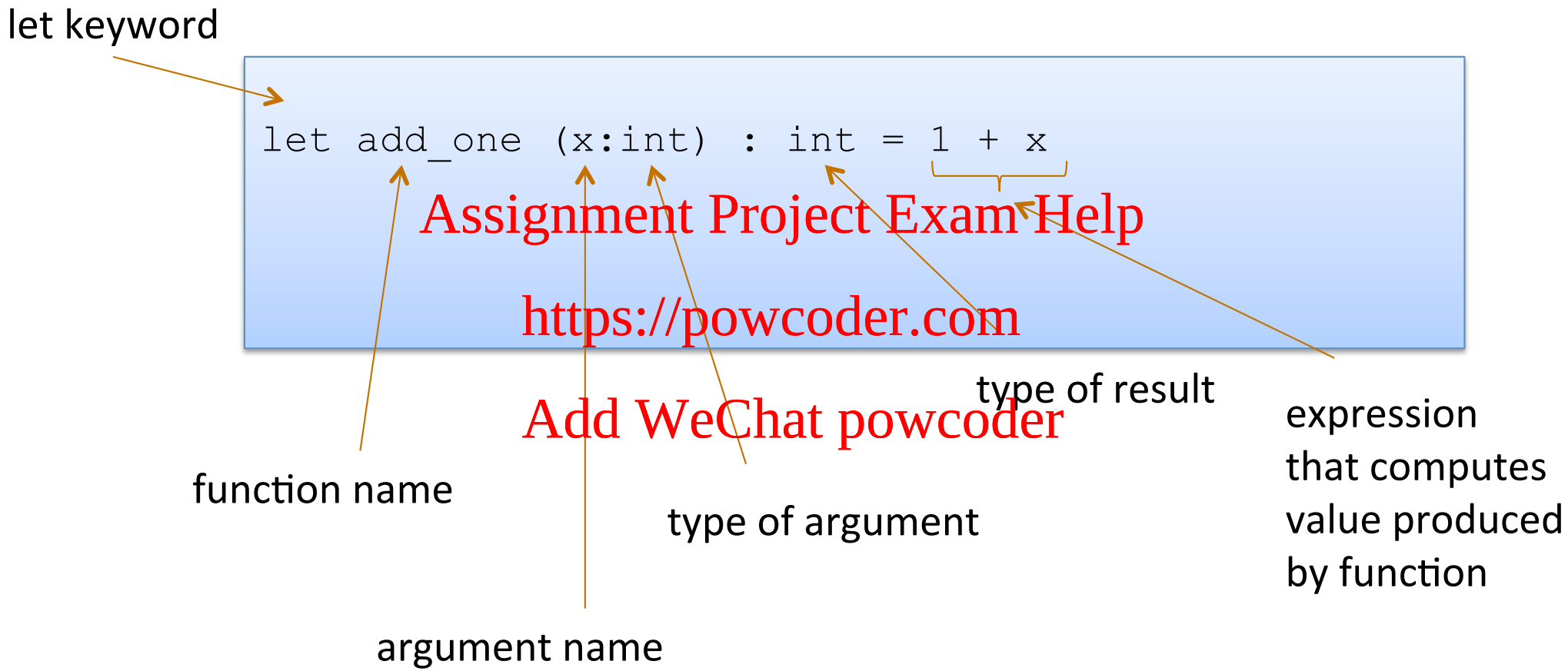
```
4 * 2
```

-->

```
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!

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

- 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
```

Learning Goals for today

- Option types
- I/O in OCaml
- Inductive data types
  Lists

Options

Options

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

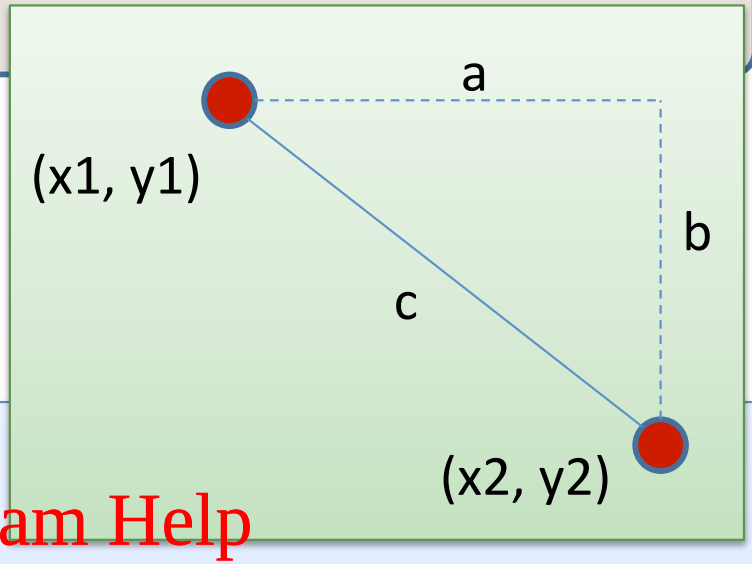
- the value None, or
- a value Some v', and v' has type t

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

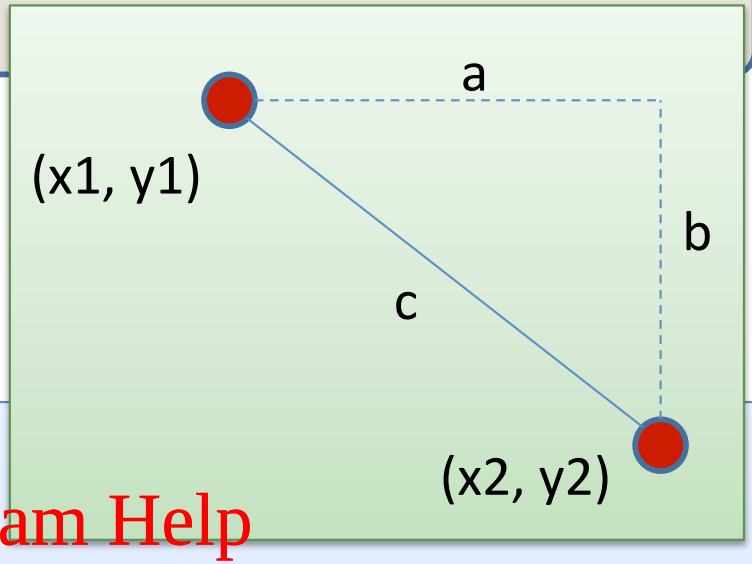
Slope between two points




```
type point = float * float

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

Slope between two points

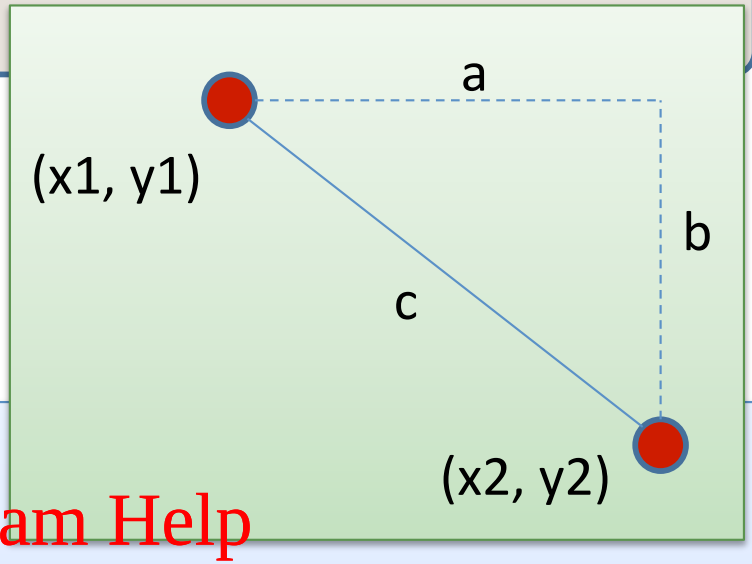



```
type point = float * float

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


deconstruct tuple

Slope between two points



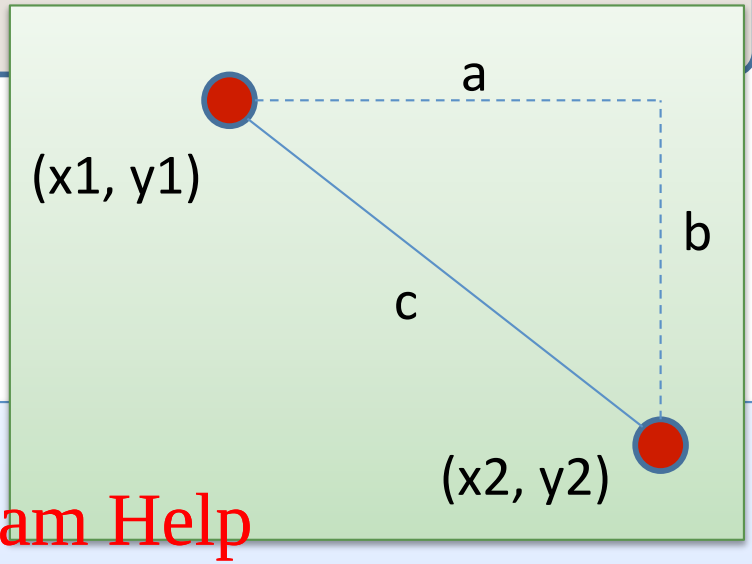

```
type point = float * float

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

avoid divide by zero

what can we return?

Slope between two points

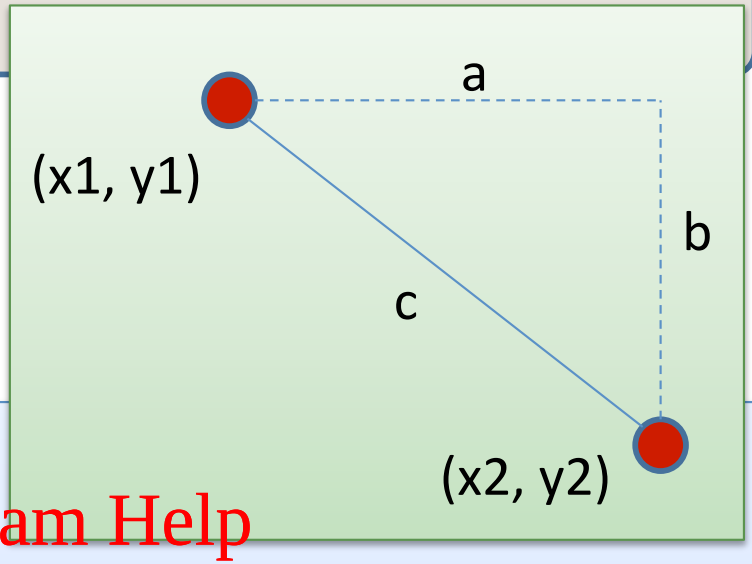



```
type point = float * float

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
    ???
  else
    ???
```

we need an option type as the result type

Slope between two points



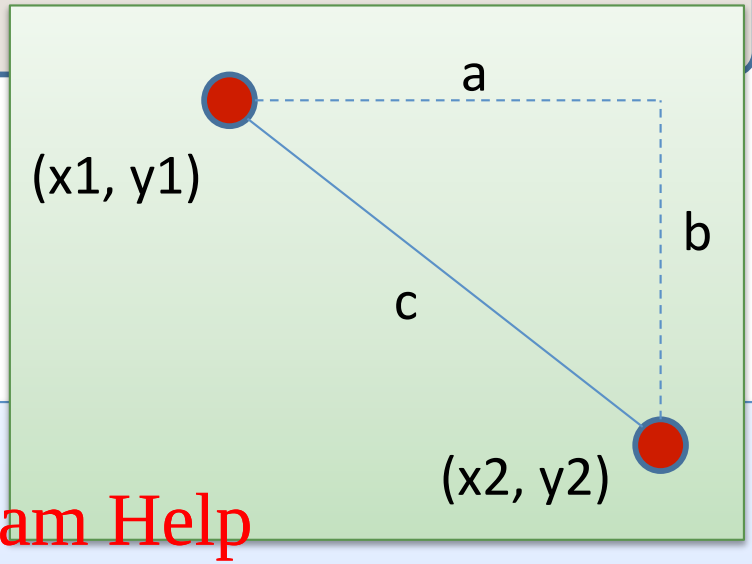

```
type point = float * float

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
    (y2 -. y1) /. xd
  else
    None
```

Has type float

Can have type float option

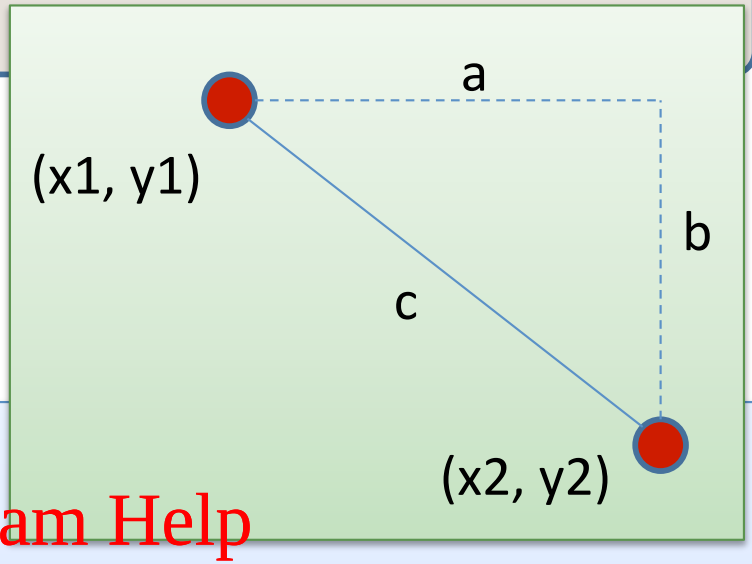
Slope between two points




```
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
```

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
```

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?

```
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?

```
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 ->
      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
```

TopHat Q1-Q5

Input/Output on files in OCaml

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.

Discarding an expression

Often we may need to discard an expression

- This happens often with `unit`, when it is returned and we don't need it.

An easy way to discard an expression is by using let with a variable that does not appear in the body:

```
let x = printf "%s/n" str in 3+2
```


In this case, we can also use underscore to avoid giving a name to this variable:

```
let _ = printf "%s/n" str in 3+2
```

This is often abbreviated in ocaml using a semicolon:

```
printf "%s/n" str; 3+2
```

Reading a line

```
let read_line (inc:in_channel) : string option =
match input_line inc with
| l -> Some l
| exception End_of_file -> None
```


Writing a line

```
Printf.fprintf ouc "%s\n" str
```

The types need to match

An example – copying one line:


```
let ic=open_in ``tmp.in'' in
let oc=open_out ``tmp.out'' in
let l=read_line ic in
let _ = match l with
      Some s -> Printf.fprintf oc ``%s/n'' s in
     |None -> ()
let _= close_in ic in
let _= close_out oc in()
```

TopHat Q6-Q7

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 values of type T from pre-existing data of type T
  - the pre-existing data is guarateed to be *smaller* than the new values

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 cases satisfy property P assuming inductive calls on *smaller* data values satisfy property P

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)

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)

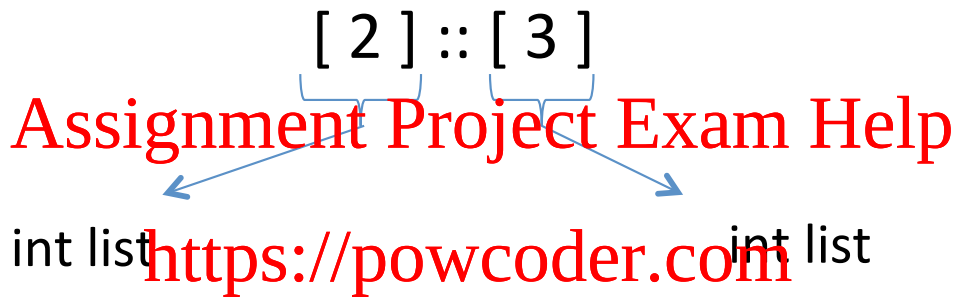
Another Example

- What type does this have?

[2] :: [3]

Another Example

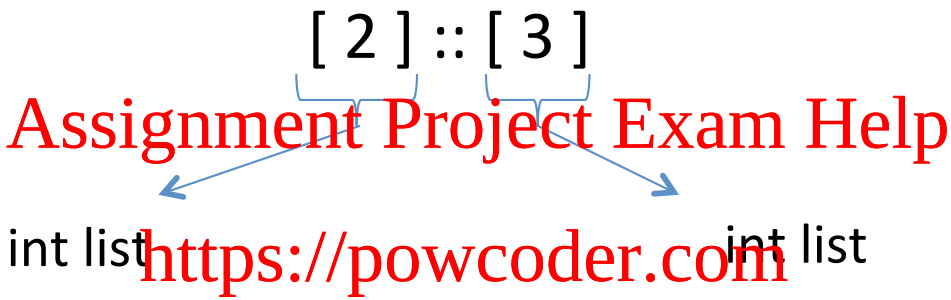
- What type does this have?




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

Another Example

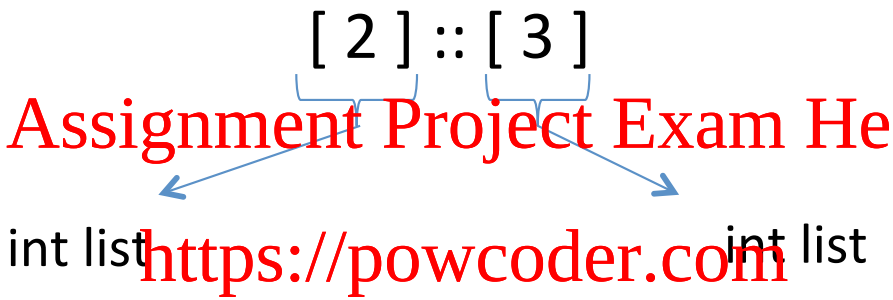
- What type does this have?



- 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

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 -> ?? :: ??
```


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
```

Three Parts to Constructing a Function

(1) Think about how to *break down* the input in to cases:

```
let rec prods (xs :
  match xs with

  | [] ->   ...

  | (x,y) :: tl ->
```

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

(2) *Assume* the recursive call on smaller data is correct.

(3) Use the result of the recursive call to *build* correct answer.

```
let rec prods (xs : (int*int) list) : int list =
  ...
  | (x,y) :: tl ->    ...  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 Q7-Q16