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

In the previous classes...

What is a Functional Language

A functional language:

- defines programs in a way similar to the one we useto define mathematical functions,
- avoids the use of mutable states (states that can change) in describing what a program should do.

In a functional language, the information is maintained by the computation.

A good functional language part of the ML family.

- Small core language,
- Supports first-class higher order functions,
- Lexically scoped
- Statically strongly typed
- Type inference
- It has a good community: ocalm.org

Terminology: Expressions, Values, Types

- Expressions are computations
  - 2 + 3 is a computation
- Values are the results of computations
  - 5 is a value
- Types describe collections of values and the computations that generate those values
  - int is a type
  - values of type int include
    - 0, 1, 2, 3, …, max_int
    - -1, -2, …, min_int

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)
5 : int (By rule 1)

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

Back to Let Expressions ... Typing

x granted type of e1 for use in e2

```
let x = e1 in
e2
```

overall expression takes on the type of e2

x has type int for use inside the let body

```
let x = 3 + 4 in
string_of_int x
```

overall expression has type string

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

Learning Goals for today

- Function Type
- Tuples
- Option Types

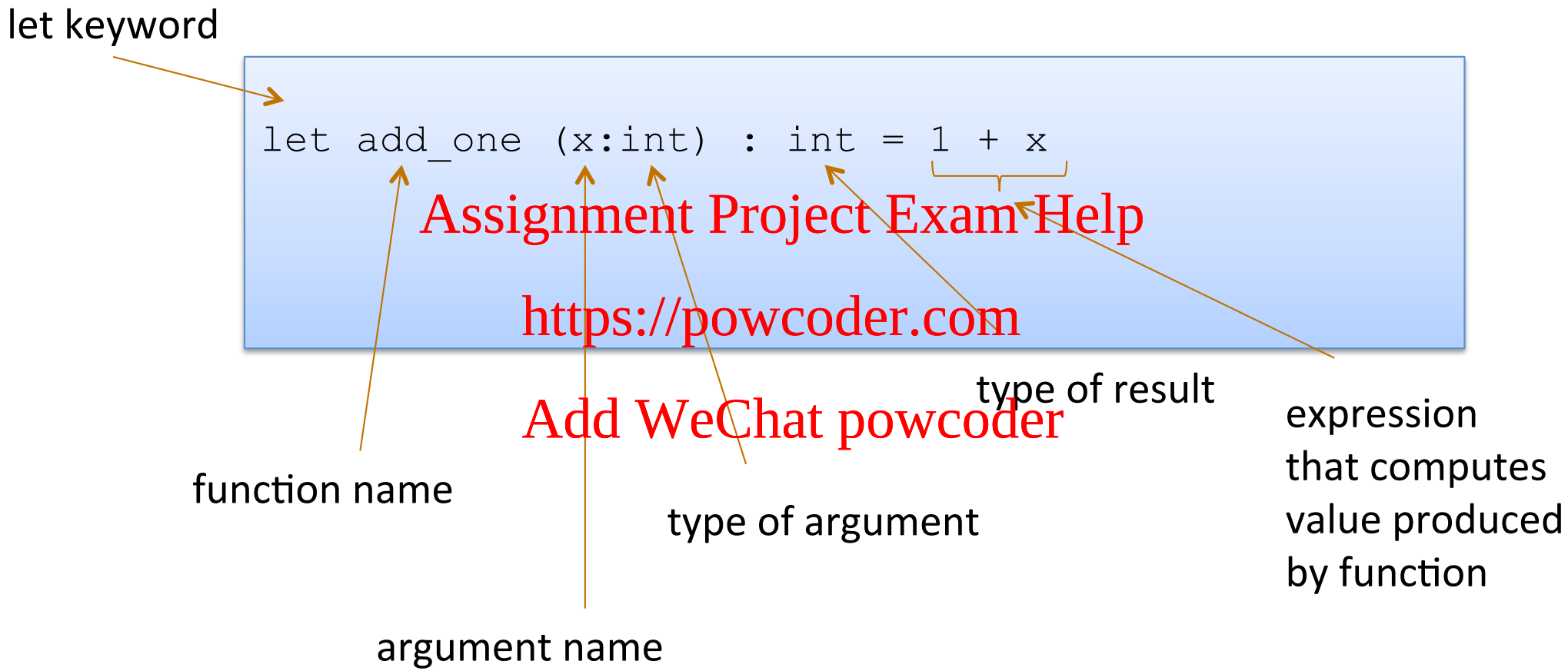
  Basic pattern matching

Functions

Defining functions

```
let add_one (x:int) : int = 1 + x
```

Defining functions



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

Defining functions

- Nonrecursive functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)
```


definition of add_one must come before use

Defining functions

- Nonrecursive functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)
```


- With a local definition:

```
let add_two' (x:int) : int =
  let add_one x = 1 + x in
  add_one (add_one x)
```

local function definition
hidden from clients

I left off the types.
O'Caml figures them out

Good style: types on
top-level definitions

Types for Functions

Some functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)

let add (x:int) (y:int) : int = x + y
```


function with two arguments

Types for functions:

```
add_one : int -> int

add_two : int -> int

add : int -> int -> int
```

Rule for type-checking functions

General Rule:

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

Example:

```
add_one : int -> int

3 + 4 : int

add_one (3 + 4) : int
```

Rule for type-checking functions

- Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```


Type:

```
add : int -> int -> int
```

Rule for type-checking functions

- Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```


Type:

```
add : int -> int -> int
```

Same as:

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

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) : ???
```

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) :
```

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
```

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

(add (3 + 4)) 7 : int
```

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

add (3 + 4) 7 : int
```

Rule for type-checking functions

Example:

```
let munge (b:bool) (x:int) : ?? =
  if not b then
    string_of_int x
  else
    "hello"
;;

let y = 17;;
```



```
munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : ??

munge true (g munge) : ??
  g : ??
```

Rule for type-checking functions

Example:

```
let munge (b:bool) (x:int) : ?? =
  if not b then
    string of int x
  else
    "hello"
;;

let y = 17;;
```



```
munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : string -> int

munge true (g munge) : ??
  g : (bool -> int -> string) -> int
```

One key thing to remember

- If you have a function f with a type like this:

A -> B -> C -> D -> E -> F

- Then each time you add an argument, you can get the type of the result by knocking off the first type in the series

f a1 : B -> C -> D -> E -> F (if a1 : A)

f a1 a2 : C -> D -> E -> F (if a2 : B)

f a1 a2 a3 : D -> E -> F (if a3 : C)

f a1 a2 a3 a4 a5 : F (if a4 : D and a5 : E)

TopHat Q1-Q6

Tuples

Tuples

- A tuple is a fixed, finite, ordered collection of values
- Some examples with their types:


```
(1, 2)                          : int * int

("hello", 7 + 3, true)          : string * int * bool

('a', ("hello", "goodbye")) : char * (string * string)
```

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
```

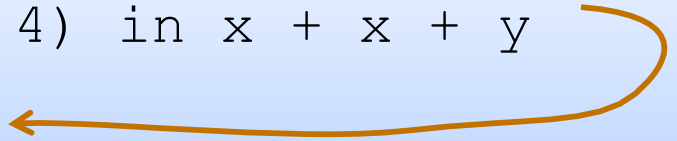
Tuples

- 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
-->  2 + 2 + 4
```



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
-->  2 + 2 + 4
-->  8
```

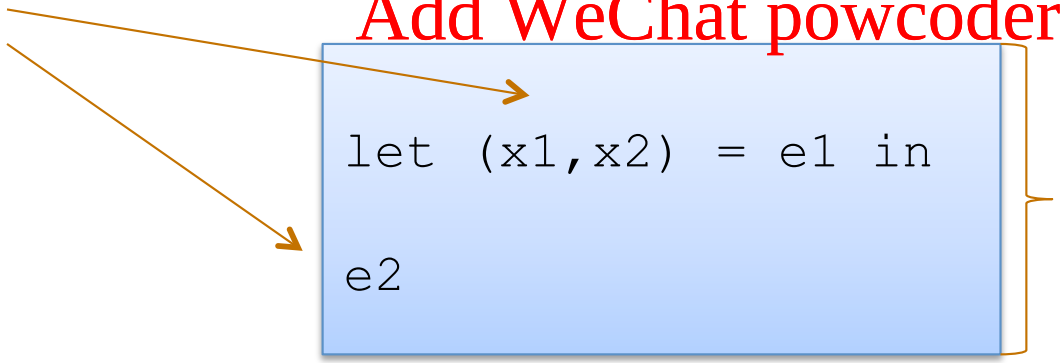
Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

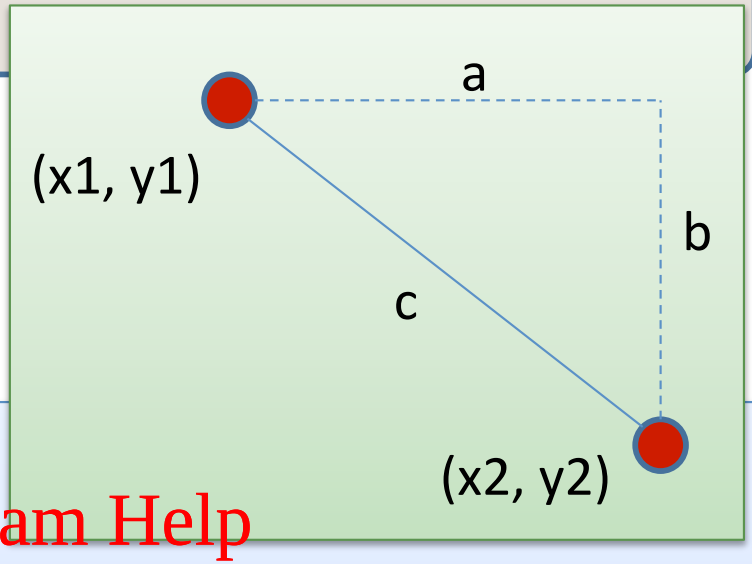
if e1 : t1 * t2 then
x1 : t1 and x2 : t2
inside the expression e2



overall expression
takes on the type of e2

Distance between two points

$c^2 = a^2 + b^2$



Problem:

- A point is represented as a pair of floating point values.
- Write a function that takes in two points as arguments and returns the distance between them as a floating point number

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)

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
   - *the argument types suggests how to do it*
5. Build new output values
   - *the result type suggests how you do it*

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
   - *the argument types suggests how to do it*
5. Build new output values
   - *the result type suggests how you do it*
6. Clean up by identifying repeated patterns
   - define and reuse helper functions
   - your code should be elegant and easy to read

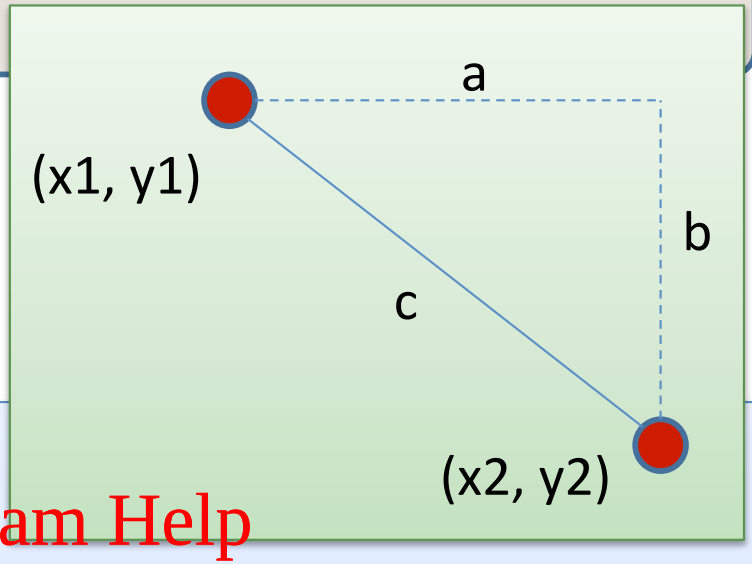
Writing Functions Over Typed Data

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
   - *the argument types suggests how to do it*
5. Build new output values
   - *the result type suggests how you do it*
6. Clean up by identifying repeated patterns
   - define and reuse helper functions
   - your code should be elegant and easy to read

Types help structure your thinking about how to write programs.

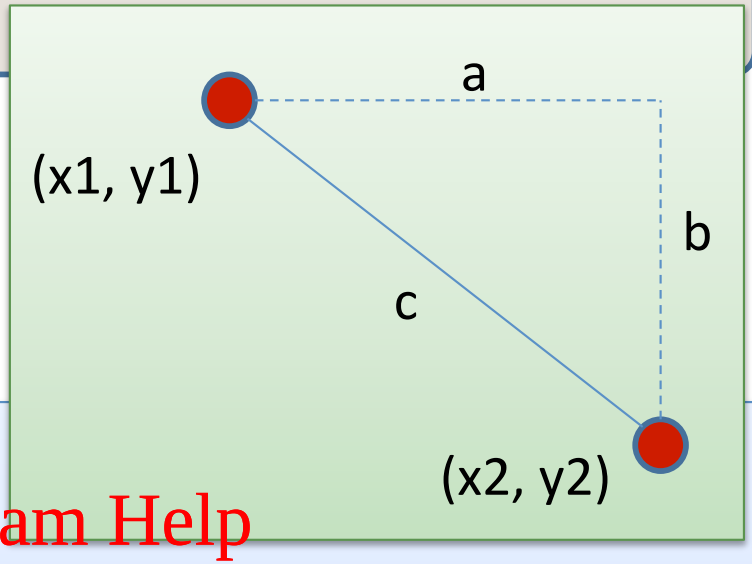
Distance between two points



a type abbreviation

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

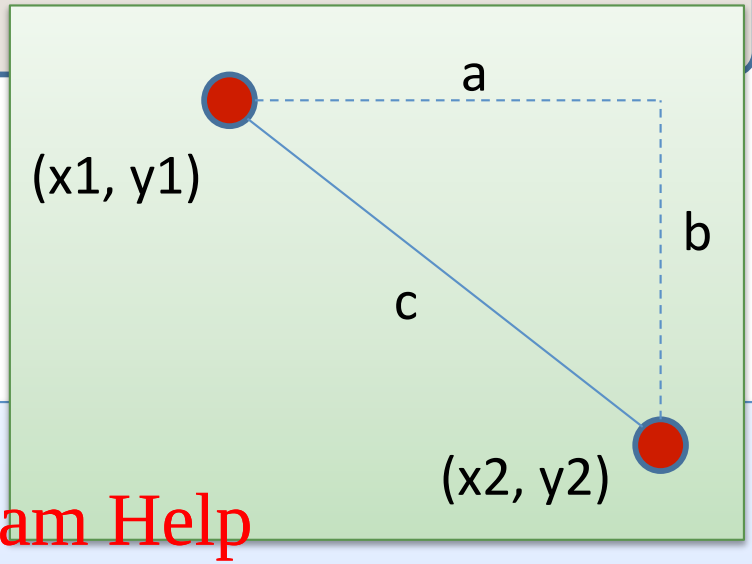
Distance between two points




```
type point = float * float

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


write down function name
argument names and types

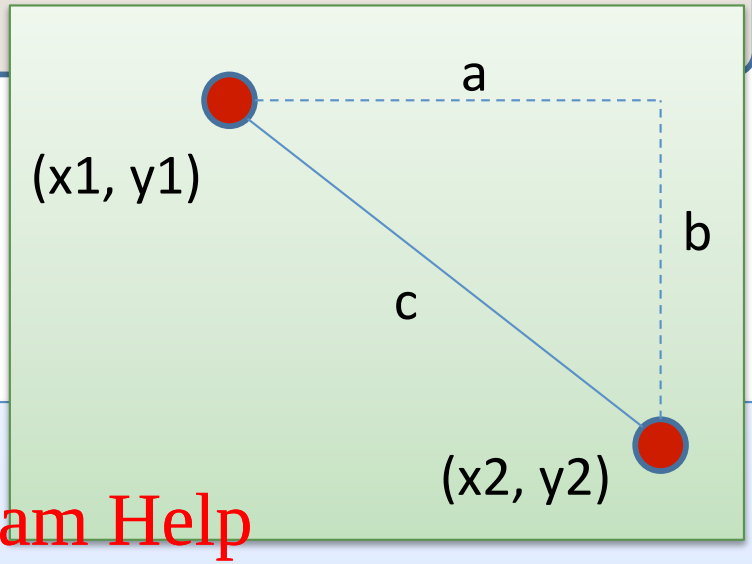
Distance between two points




```
type point = float * float

let distance (p1:point) (p2:point) : float =

  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  ...
```


deconstruct
function inputs

Distance between two points

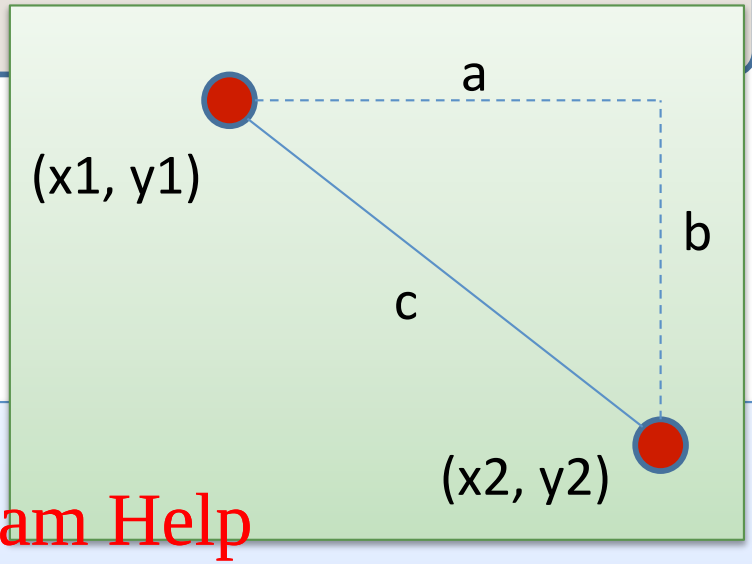



```
type point = float * float

let distance (p1:point) (p2:point) : float =

  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt ((x2 -. x1) *. (x2 -. x1) +.
        (y2 -. y1) *. (y2 -. y1))
```

compute function results

notice operators on floats have a "." in them

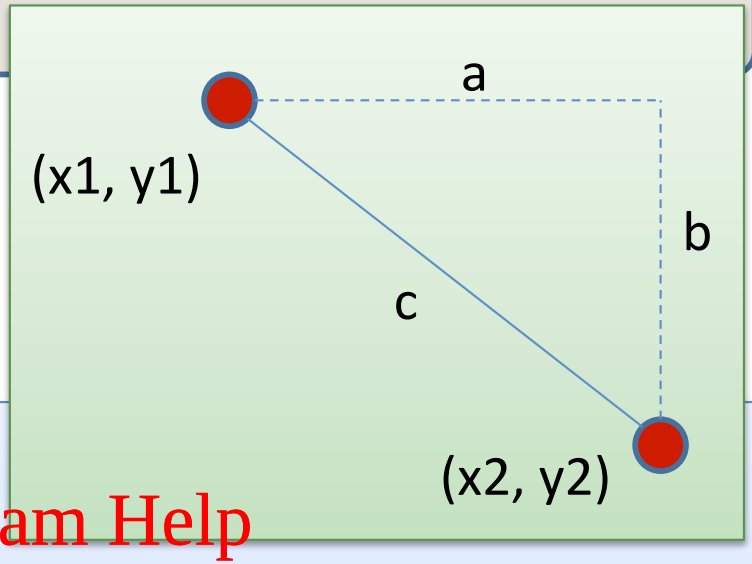
Distance between two points




```
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))
```


define helper functions to avoid repeated code

Distance between two points




```
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))


let pt1 = (2.0,3.0)
let pt2 = (0.0,1.0)
let dist12 = distance pt1 pt2
```


testing

TopHat Q7-Q10

Unit

Tuples

- Here's a tuple with 2 fields:

(4.0, 5.0) : float * float

- Here's a tuple with 3 fields:

(4.0, 5, "hello") : float * int * string

- Here's a tuple with 4 fields:

(4.0, 5, "hello", 55) : float * int * string * int

Tuples

- Here's a tuple with 2 fields:

(4.0, 5.0) : float * float

- Here's a tuple with 3 fields:

(4.0, 5, "hello") : float * int * string

- Here's a tuple with 4 fields:

(4.0, 5, "hello", 55) : float * int * string * int

- Here's a tuple with 0 fields:

() : unit

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

TopHat Q11-Q12

Options

Options

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

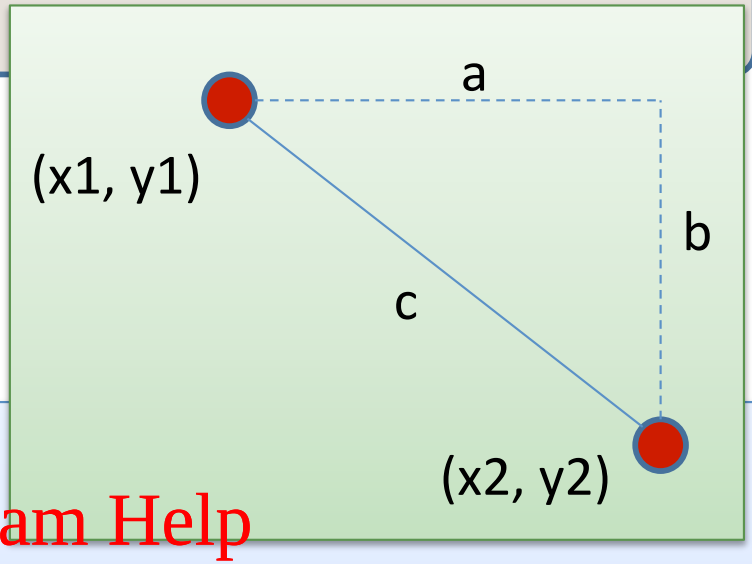
- the value None, or
- a value Some v', and v' has type t

Options can signal there is no useful result to the computation

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

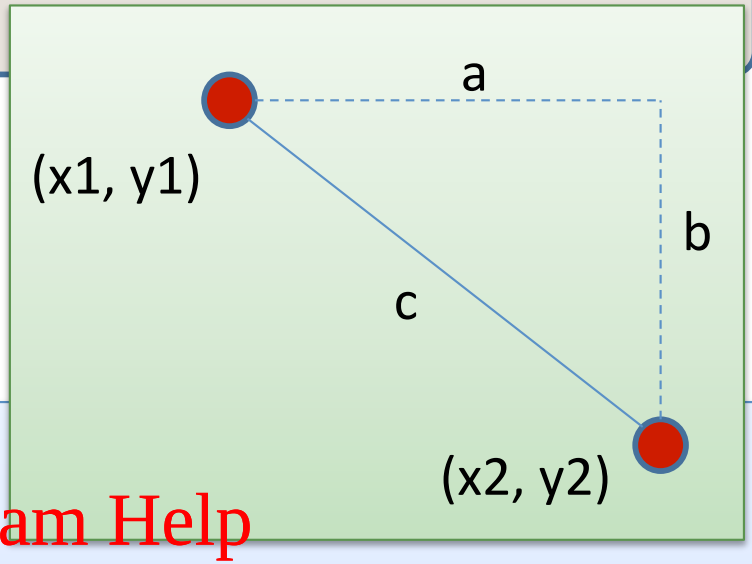
- If the key is present, return Some v where v is the associated value
- If the key is not present, we return None

Slope between two points




```
type point = float * float

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

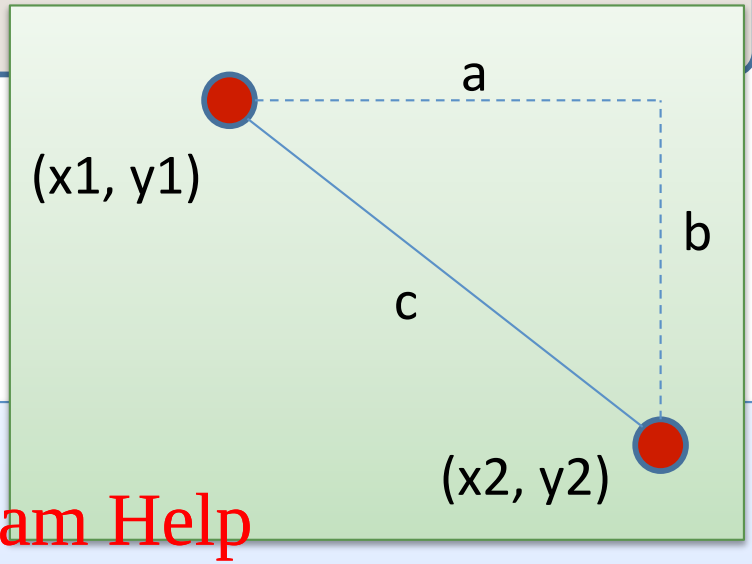
Slope between two points




```
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

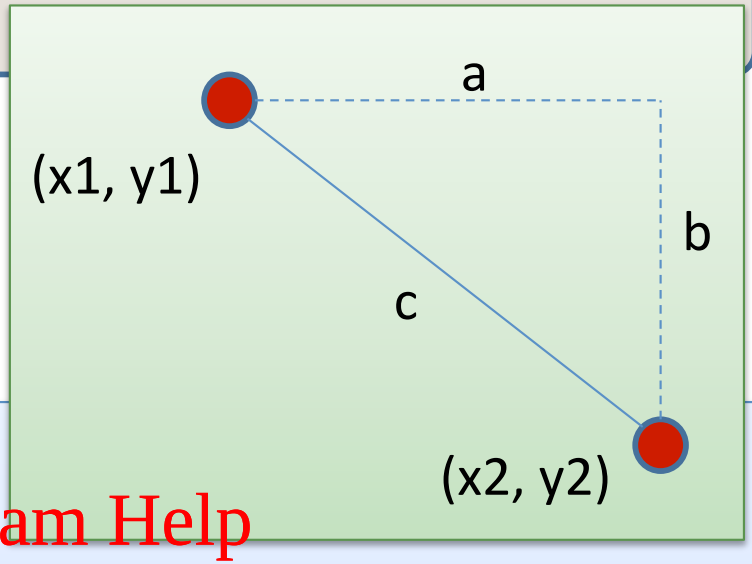



```
type point = float * float

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?

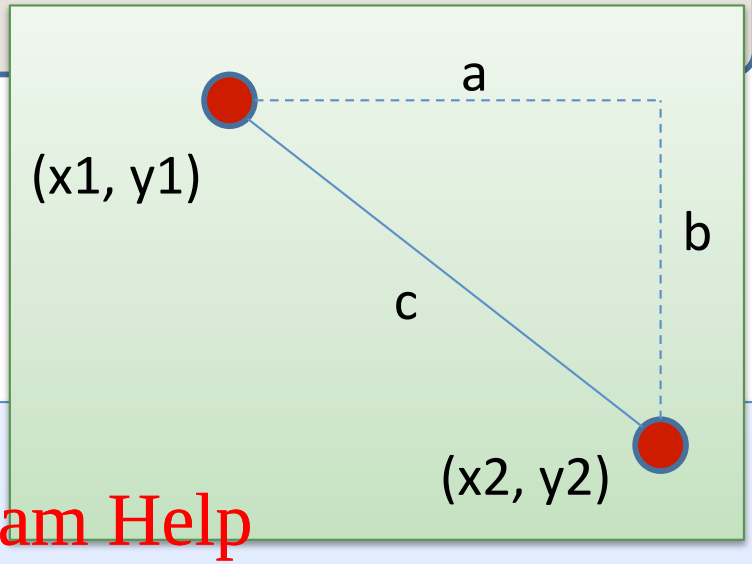
Slope between two points




```
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
    ???
```

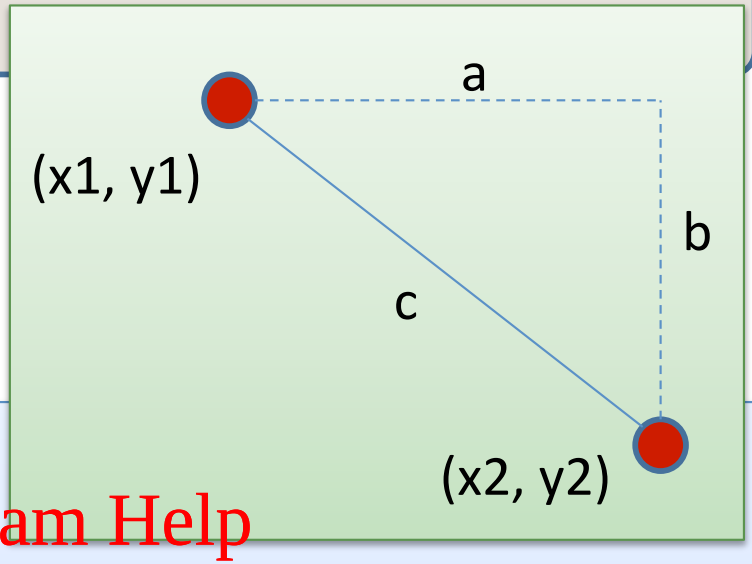
we need an option type as the result type

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

Slope between two points

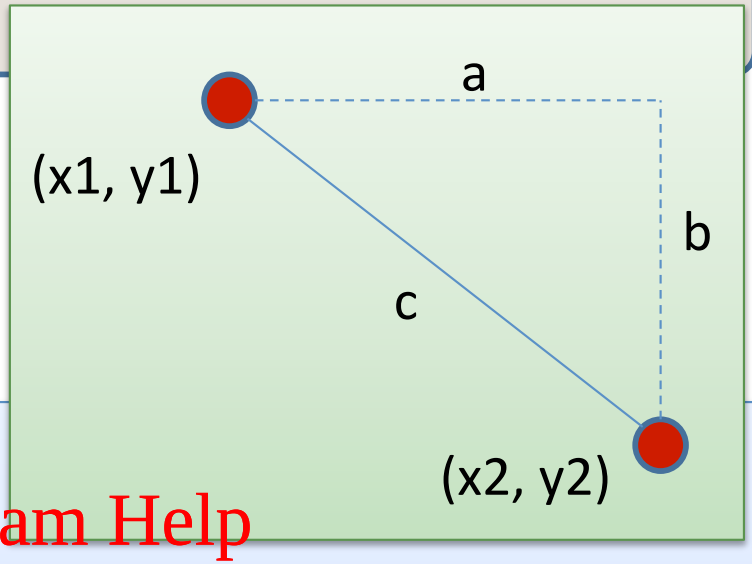



```
type point = float * float

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

Has type float

Can have type float option

Slope between two points



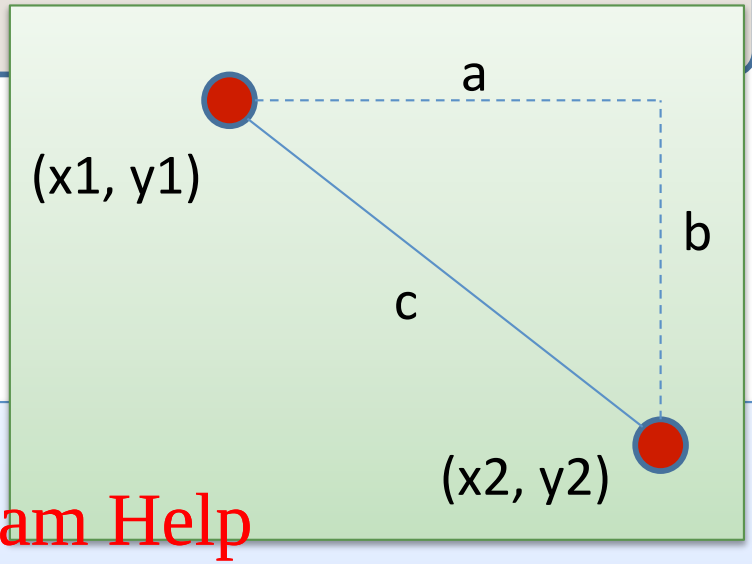

```
type point = float * float

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

Has type float

Can have type float option

WRONG: Type mismatch

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

doubly WRONG:
result does not
match declared result

Remember the typing rule for if

if e1 : bool
and e2 : t and e3 : t (for some type t)
then if e1 then e2 else e3 : t

Returning an optional value from an if statement:


```
if … then

   None              : t option

else

   Some ( … )        : t 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?

```
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
```

Summary

- Function Type
- Tuples
- Option Types

Basic pattern matching