# O’Caml Basics: Unit and Options

COS 326
David Walker
Princeton University

# Tuples

- Here's a tuple with 2 fields:

(4.0, 5.0) : float * float

# 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
```

# 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

- Have you ever thought about what a  tuple with 0 fields might look like?

# Unit

- Unit is the tuple with zero fields!

```
() : unit
```

- 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!

() : unit

- 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!

() : unit

- 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 tuples:
  - when the input has type unit
    - use let () = … in … to deconstruct
    - or better use e1; … to deconstruct if e1 has type unit
    - or do nothing … because unit carries no information of value
  - when the output has type unit
    - use () to construct

# OUR THIRD DATA STRUCTURE!
## THE OPTION

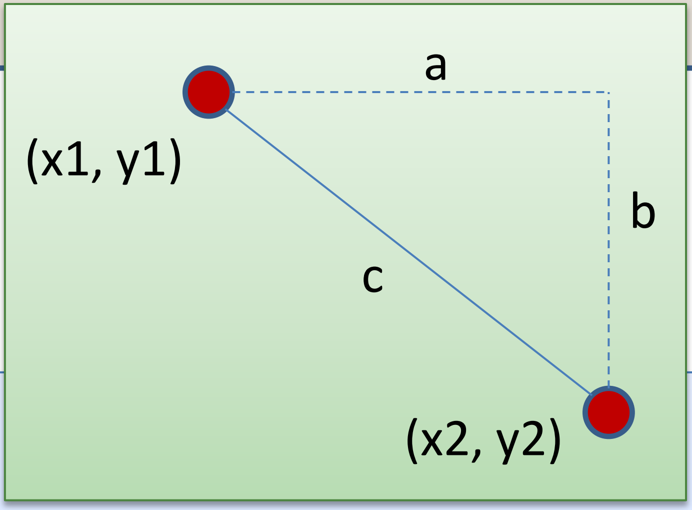
# Options

- 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

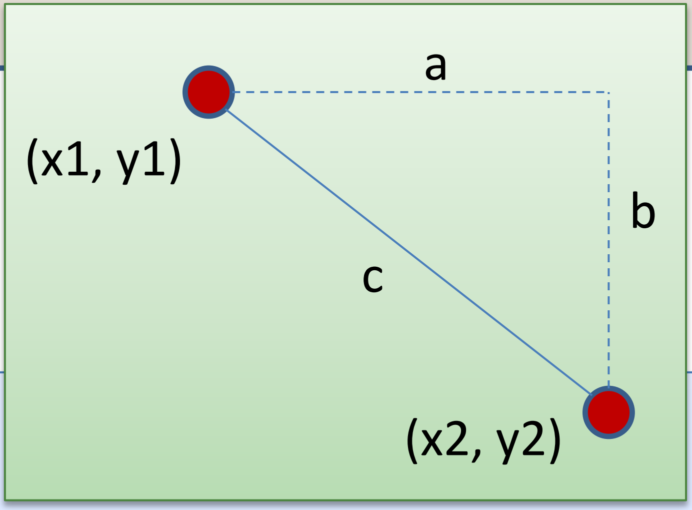
- Example: we loop up a value in a hash table using a key.
  - If the key is present in the hash table then we return Some v where v is the associated value
  - If the key is not present, we return None

# Slope between two points




```
type point = float * float

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



;;
```

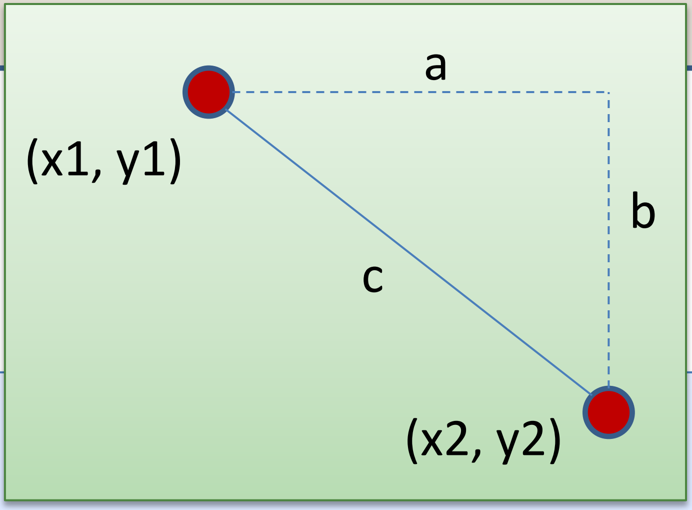
# Slope between two points




```
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

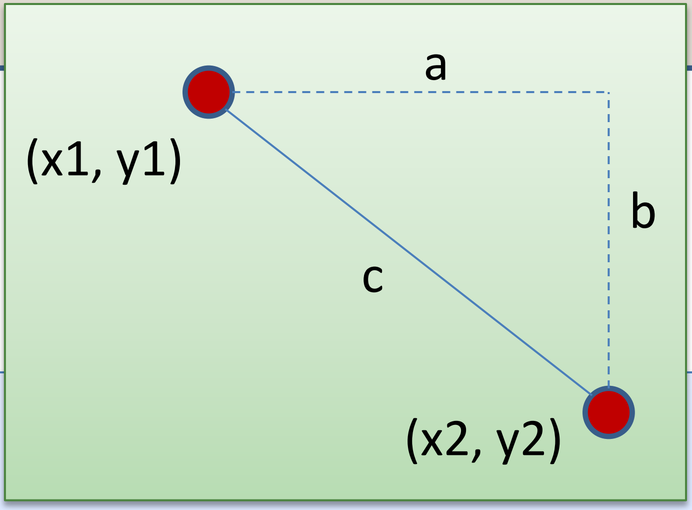



```
type point = float * float

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?

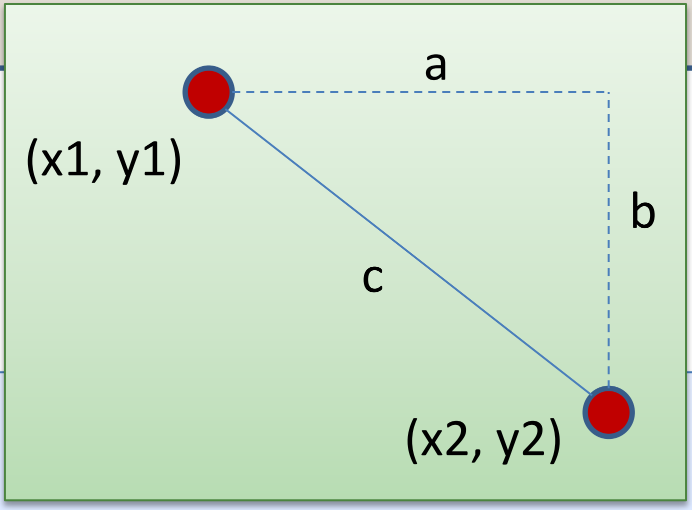
# Slope between two points




```
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
    ???
;;
```

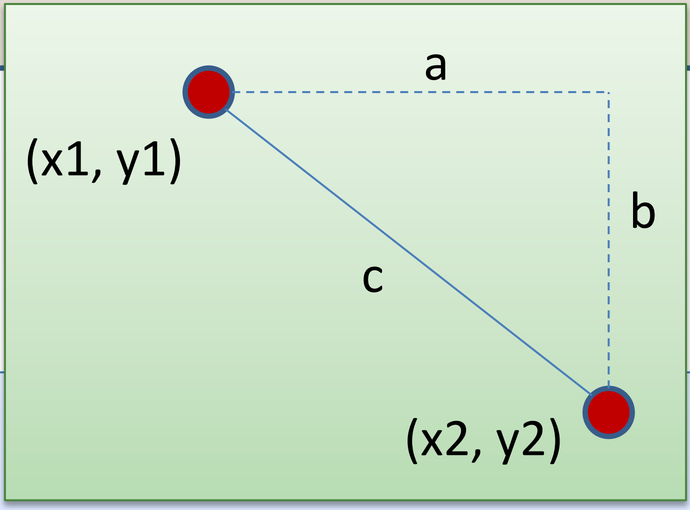
we need an option type as the result type

# 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

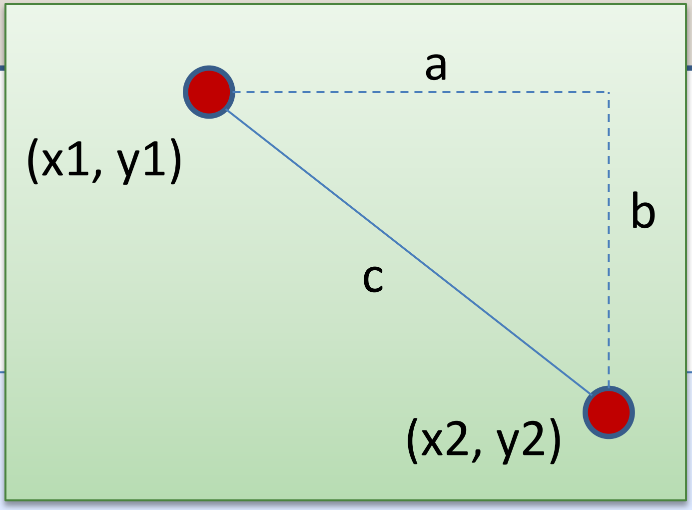



```
type point = float * float

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



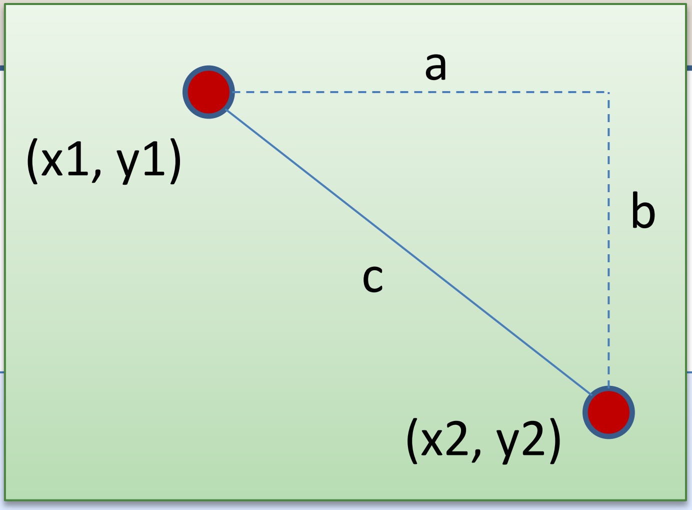

```
type point = float * float

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

WRONG: Type mismatch

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

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 tuples:

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
```

# MORE PATTERN MATCHING

# Recall the Distance Function

```
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))
;;
```

# Recall the Distance Function

```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match p1 with
  | (x1,y1) ->
       let (x2,y2) = p2 in
       sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;
```

There is only 1 possibility when matching a pair

# Recall the Distance Function

```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match p1 with
  | (x1,y1) ->
     match p2 with
     | (x2,y2) ->
        sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;
```

We can nest one match expression inside another.
(We can nest any expression inside any other, if the expressions have the right types)

# Better Style: Complex Patterns

we built a pair of pairs

```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match (p1, p2) with
  | ((x1,y1), (x2, y2)) ->
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;
```

Pattern for a pair of pairs: ((variable, variable), (variable, variable))
All the variable names in the pattern must be different.

# Better Style: Complex Patterns

we built a pair of pairs

```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match (p1, p2) with
  | (p3, p4) ->
    let (x1, y1) = p3 in
    let (x2, y2) = p4 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;
```

A pattern must be consistent with the type of the expression
in between match … with
We use (p3, p4) here instead of ((x1, y1), (x2, y2))

# I like the original the best

```
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))
;;
```

It is the clearest and most compact.
Code with unnecessary nested patterns matching is particularly ugly to read.
You'll be judged on code style in this class.

# Combining patterns

```
type point = float * float

(* returns a nearby point in the graph if one exists *)
nearby : graph -> point -> point option

let printer (g:graph) (p:point) : unit =
  match nearby g p with
  | None -> print_string "could not find one\n"
  | Some (x,y) ->
      print_float x;
      print_string ", ";
      print_float y;
      print_newline();
;;
```

# Other Patterns

- Constant values can be used as patterns

```
let small_prime (n:int) : bool =
  match n with
  | 2 -> true
  | 3 -> true
  | 5 -> true
  | _ -> false
;;
```

```
let iffy (b:bool) : int =
  match b with
  | true -> 0
  | false -> 1
;;
```

the underscore pattern
matches anything
it is the "don't care" pattern

# A QUICK COMMENT ON JAVA

# 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?

# A Paucity of Types

```
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?*

# From Java Pairs to O'Caml Pairs

In O'Caml, if a pair may be null it is a pair option:

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

# From Java Pairs to O'Caml Pairs

In O'Caml, if a pair may be null it is a pair option:

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

If you write code like this:

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

# From Java Pairs to O'Caml Pairs

In O'Caml, if a pair may be null it is a pair option:

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

If you write code like this:

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

The type checker gives you an error immediately:

```
# … 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 O'Caml Pairs

In O'Caml, if a pair may be null it is a pair option:

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

What if you did the following stupid thing?

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

# From Java Pairs to O'Caml Pairs

In O'Caml, if a pair may be null it is a pair option:

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

What if you did the following stupid thing?

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

The type checker to the rescue again:

```
 ..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 O'Caml Pairs

In O'Caml, if a pair may be null it is a pair option:

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

You can fix either error in 2 seconds:

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

```
let swap_java_pair (p:java_pair) : java_pair =
   match p with
     | None -> None
     | Some (x,y) -> Some (y,x)
```

# From Java Pairs to O'Caml Pairs

- Moreover, your pairs are probably almost never null
- Defensive programming in which you are always checking for null is annoying and time consuming
- Worst of all, there just isn't always some "good thing" for a function to do when it receives a bad input, like a null pointer
- In O'Caml, all these issues disappear when you use the proper type for a pair and that type contains no "extra junk"

```
type pair = int * int
```

```
let swap (p:pair) : pair =
   let (x,y) = p in (y,x)
```

- Once you know O'Caml, it is *hard* to write swap incorrectly

# Summary of Java Pair Rant

- 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 …
  - Later: there is no type to describe just the acyclic lists or binary trees …
- O'Caml 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
  - type checking and pattern analysis help prevent programmers from ever forgetting about a case

# OVERALL SUMMARY:
# A SHORT INTRODUCTION TO FUNCTIONAL PROGRAMMING

# Functional Programming

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
4. Deconstruct input data structures
   - the argument types suggest how you do it
   - the types tell you which cases you must cover
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
   - refactor code to use your helpers
   - your code should be elegant and easy to read

# Summary: Constructing/Deconstructing Values

<table>
<tr><th>Type</th><th>Construct Values</th><th>Number of Cases</th><th>Deconstruct Values</th></tr>
<tr><td>int</td><td>0, -1, 2, …</td><td>2^31-1</td><td>match i with<br>| 0 -> …<br>| -1 -> …<br>…<br>| x -> …</td></tr>
<tr><td>bool</td><td>true, false</td><td>2</td><td>match b with<br>| true -> …<br>| false -> ….</td></tr>
<tr><td>t1 * t2</td><td>(2, "hi")</td><td>(# of t1) * (# of t2)</td><td>let (x,y) = … in …<br><br>match p with (x,y) -> …</td></tr>
<tr><td>unit</td><td>()</td><td>1</td><td>e1; …</td></tr>
<tr><td>t option</td><td>None, Some 3</td><td>1 + (# of t1)</td><td>match opt with<br>| None -> …<br>| Some x -> …</td></tr>
</table>

# END