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 unit type:
  - 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:

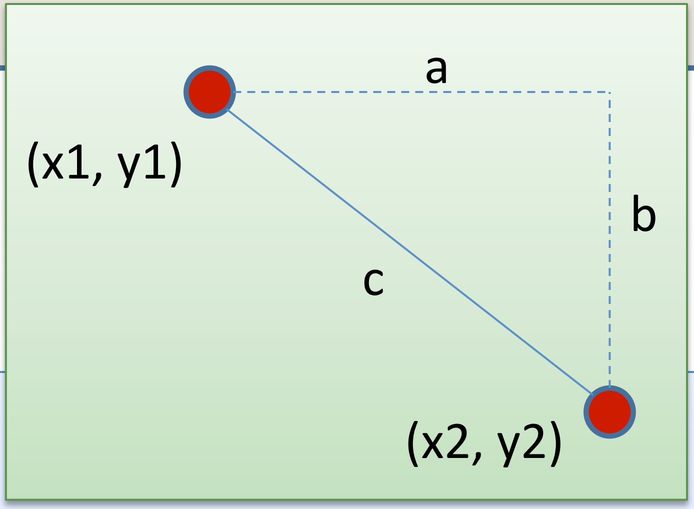
- the value None, or
- a value Some v', and v' has type t

Options can signal there is no useful result to the computation

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

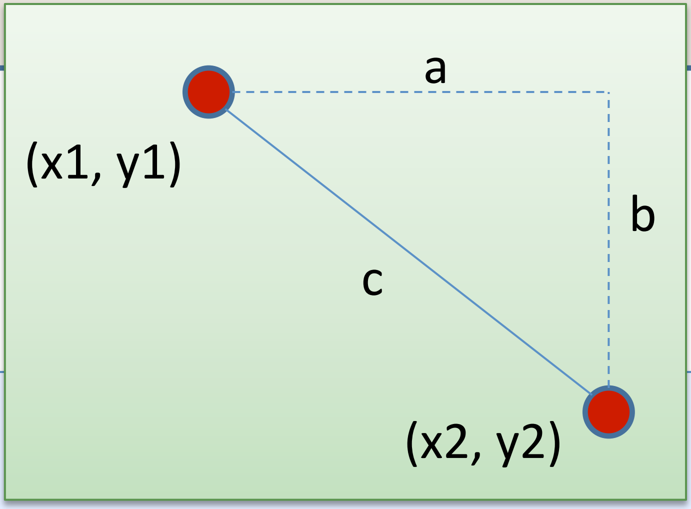
- If the key is present, return Some v where v is the associated value
- If the key is not present, we return None

Slope between two points



```
type point = float * float

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



;;
```

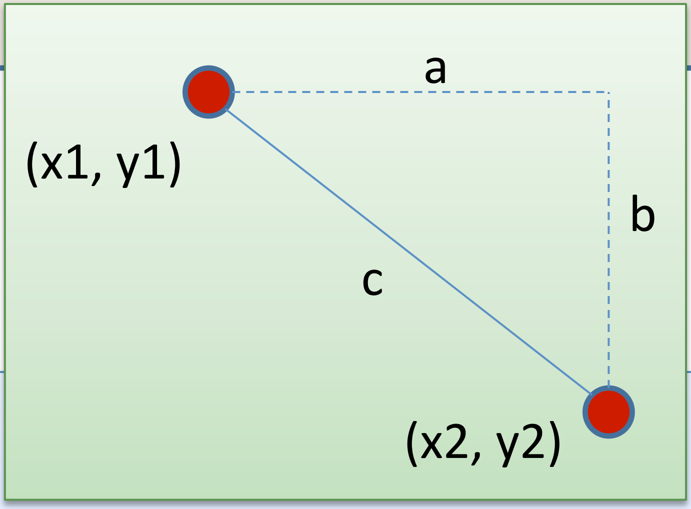
Slope between two points



```
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

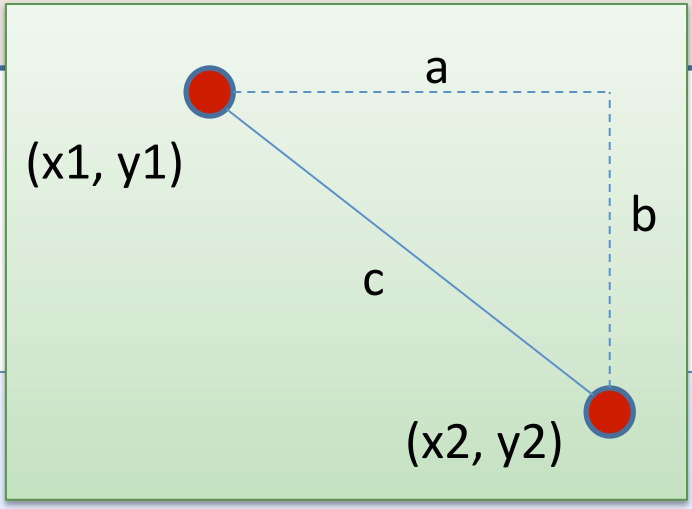


```
type point = float * float

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?

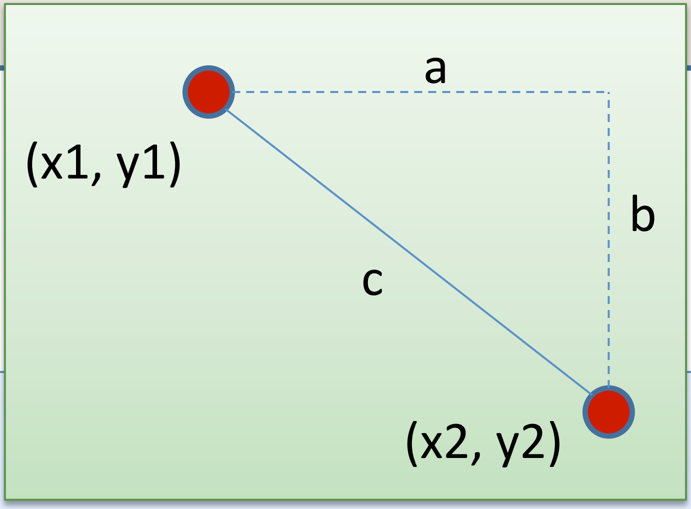
Slope between two points



```
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
    ???
;;
```

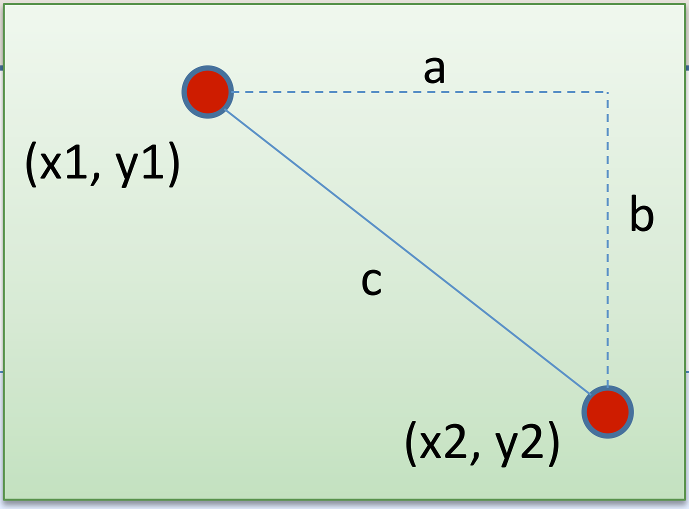
we need an option
type as the result type

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

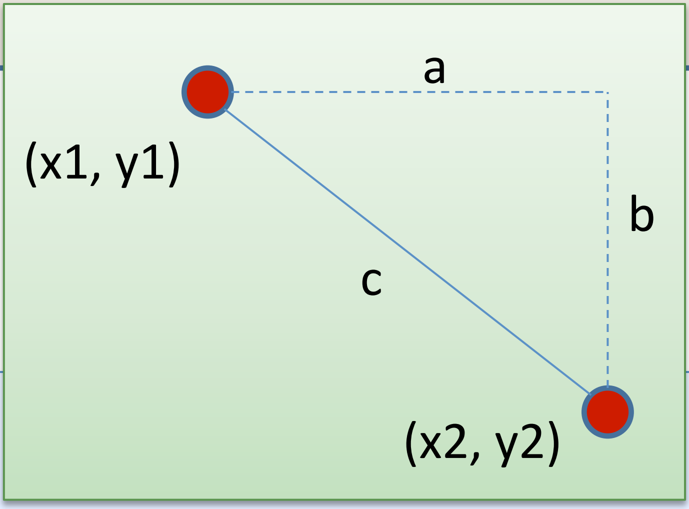


```
type point = float * float

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



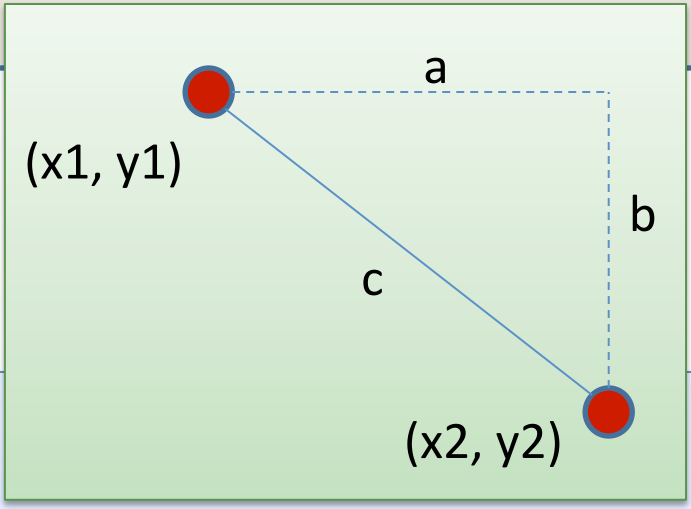
```
type point = float * float

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
```

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

(x2, y2) is an example of a pattern – a pattern for tuples.

So let declarations can contain patterns just like match statements

The difference is that a match allows you to consider multiple different data shapes

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
;;
```

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

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

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

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

END