Options

Speaker: David Walker
COS 326
Princeton University
Often, we either have a thing .... or we don’t:

17  “hi”
Often, we either have a thing .... or we don’t:

Option types are used in this situation: `t option`
Often, we either have a thing .... or we don’t:

```
17  “hi”
```

Option types are used in this situation:  \( t \) option

There’s \textit{one way} to build a pair, but \textit{two ways} to build an optional value:

- \textbf{None} -- when we’ve got nothing
- \textbf{Some} \( v \) -- when we’ve got a value \( v \) of type \( t \)
Slope between two points

type point = float * float

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

```plaintext
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
```
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
Slope between two points

`type point = float * float`

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

```plaintext
let (x1, y1) = p1 in
let (x2, y2) = p2 in
let xd = x2 -. x1 in
if xd != 0.0 then
    (y2 -. y1) /.
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

\[
\text{if } e_1 : \text{bool} \\
\text{and } e_2 : t \text{ and } e_3 : t \text{ (for some type } t) \\
\text{then if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\]

Returning an optional value from an if statement:

\[
\text{if ... then} \\
\quad \text{None} : t \text{ option} \\
\text{else} \\
\quad \text{Some ( ... )} : t \text{ option}
\]
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?

\[ \text{slope : point} \rightarrow \text{point} \rightarrow \text{float option} \]

\[ \text{let print_slope (p1:point) (p2:point) : unit = match slope p1 p2 with} \]
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`

```ocaml
define 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 -> |
  | None -> |

You can put a “|” on the first line if you want. It is generally considered better style to do so.
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

```ocaml
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

```ocaml
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

A pattern must be **consistent with** the type of the expression between `match ... with`

We use `(p3, p4)` here instead of `(((x1, y1), (x2, y2))`
type point = float * float

let distance ((x1, y1):point) ((x2, y2):point) : float =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))

Function parameters are patterns too!
What’s the best style?

```ocaml
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))
```

```ocaml
let distance ((x1,y1):point) ((x2,y2):point) : float =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

Either of these is reasonably clear and compact. Code with unnecessary nested matches/lets is particularly ugly to read. You'll be judged on code style in this class.
What’s the best style?

This is how I'd do it ... the types for tuples + the tuple patterns are a little ugly/verbose ... but for now in class, use the explicit type annotations. We will loosen things up later in the semester.

```
let distance (x1,y1) (x2,y2) =
    let square x = x *. x in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```
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
Exercises

Exercise 1: What is the type of `foo` below? Of `bar`? (bar is used but isn’t shown)

```ocaml
let foo (a, b, c) d = 
  match bar a with
  | (_, Some x) -> if x then None else Some d
  | ((x, y), None) -> if a + b < 17 then Some (x ^ "hi") else Some y
```

Exercise 2: Consider these two types:

```ocaml
type t = (bool * bool) option
type s = (bool option) * (bool option)
```

Do they contain the same “amount” of information?
Write a function to convert data with type `t` to type `s`.
And another function to convert data with type `s` back to type `t`.
What happens?
Explain when a program you write might use `s` instead of `t` and vice versa.