Simple Data

COS 326
Andrew W. Appel
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
What is the single most important mathematical concept ever developed in human history?
What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable
What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

(runner up: natural numbers/induction)
Why is the mathematical variable so important?

The mathematician says:

“Let $x$ be some integer, we define a polynomial over $x$ ...”
Why is the mathematical variable so important?

The mathematician says:

“Let x be some integer, we define a polynomial over x ...”

What is going on here? The mathematician has separated a definition (of x) from its use (in the polynomial).

This is the most primitive kind of abstraction (x is some integer).

Abstraction is the key to controlling complexity and without it, modern mathematics, science, and computation would not exist.
OCAML BASICS:
LET DECLARATIONS
• Good programmers identify repeated patterns in their code and factor out the repetition into meaningful components
• In O’Caml, the most basic technique for factoring your code is to use let expressions
• Instead of writing this expression:

\[(2 + 3) \times (2 + 3)\]
Abstraction & Abbreviation

• Good programmers identify repeated patterns in their code and factor out the repetition into meaning components.

• In O’Caml, the most basic technique for factoring your code is to use *let* expressions.

• Instead of writing this expression:

\[(2 + 3) \times (2 + 3)\]

• We write this one:

```
let x = 2 + 3 in
x * x
```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
A Few More Let Expressions

let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs

let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
Abstraction & Abbreviation

- Two kinds of let:

```
if tuesday() then
    let x = 2 + 3 in
    x + x
else
    0
;;
```

```
let x = 2 + 3 ;;
let y = x + 17 / x ;;
```

- `let ... in ...` is an *expression* that can appear inside any other *expression*

  The scope of x does not extend outside the enclosing “in”

- `let ... ;;` without “in” is a top-level *declaration*

  Variables x and y may be exported; used by other modules

  (Don’t need ;; if another let comes next; do need it if the next top-level declaration is an expression)
• Each OCaml variable is *bound* to 1 value
• *The value to which a variable is bound to never changes!*

```ocaml
let x = 3 ;;

let add_three (y:int) : int = y + x ;;
```
• Each OCaml variable is **bound** to 1 value
• *The value to which a variable is bound to never changes!*

```ocaml
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
```

*It does not matter what I write next. add_three will always add 3!*
Binding Variables to Values

- Each OCaml variable is bound to 1 value
- *The value a variable is bound to never changes!*

```ocaml
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
let x = 4 ;;
let add_four (y:int) : int = y + x ;;
```

A distinct variable that "happens to be spelled the same"
Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them.

```ocaml
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
let zzz = 4 ;;
let add_four (y:int) : int = y + zzz ;;
let add_seven (y:int) : int =
  add_three (add_four y) ;;
```

rename x to zzz if you want to, replacing its uses.
Binding Variables to Values

- Each OCaml variable is bound to 1 value
- OCaml is a **statically scoped** language

```ocaml
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
let x = 4 ;;
let add_four (y:int) : int = y + x ;;
let add_seven (y:int) : int = add_three (add_four y) ;;
```
How do let expressions operate?

```plaintext
let x = 2 + 1 in x * x
```
How do let expressions operate?

```plaintext
let x = 2 + 1 in x * x
```

--> 

```plaintext
let x = 3 in x * x
```
How do let expressions operate?

\[
\text{let } x = 2 + 1 \text{ in } x \times x
\]

\[
\rightarrow
\]

\[
\text{let } x = 3 \text{ in } x \times x
\]

\[
\rightarrow
\]

\[
3 \times 3
\]

substitute 3 for x
How do let expressions operate?

```
let x = 2 + 1 in x * x
```

--> 

```
let x = 3 in x * x
```

--> 

```
3 * 3
```

--> 

```
9
```

substitute 3 for x
How do let expressions operate?

```
let x = 2 + 1 in x * x
```

```
-->
```

```
let x = 3 in x * x
```

```
-->
```

```
3 * 3
```

```
-->
```

```
9
```

**Note:** I write $e_1 \rightarrow e_2$ when $e_1$ evaluates to $e_2$ in one step.
Did you see what I did there?
Did you see what I did there?

I defined the language in terms of itself:

\[
\text{let } x = 2 \text{ in } x + 3 \quad \rightarrow \quad 2 + 3
\]

I’m trying to train you to think at a high level of abstraction.

*I didn’t have to mention low-level abstractions like assembly code or registers or memory layout*
Another Example

let x = 2 in
let y = x + x in
y * x
Another Example

```
let x = 2 in
let y = x + x in
y * x
```

```
let x = 2 in
let y = x + x in
y * x
```

```
let y = 2 + 2 in
y * 2
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```
Another Example

\[
\text{let } x = 2 \text{ in } \\
\text{let } y = x + x \text{ in } \\
y * x
\]

\[
\text{substitute } 2 \text{ for } x
\]

\[
\text{let } y = 2 + 2 \text{ in } \\
y * 2
\]

\[
\text{let } y = 4 \text{ in } \\
y * 2
\]
Another Example

```
let x = 2 in
let y = x + x in
y * x

-->
let y = 2 + 2 in
y * 2

-->
let y = 4     in
y * 2

-->
4 * 2
```

substitute 2 for x

substitute 4 for y
Another Example

let \( x = 2 \) in
let \( y = x + x \) in
\( y \times x \)

\( \rightarrow \)

let \( y = 2 + 2 \) in
\( y \times 2 \)

\( \rightarrow \)

let \( y = 4 \) in
\( y \times 2 \)

\( \rightarrow \)

\( 4 \times 2 \)

\( \rightarrow \)

8

Moral: Let operates by substituting computed values for variables
What would happen in an imperative language?

C program:

```c
x = 2;
x += x;
return x*2;
```

->

```c
x += 2  ???
return x*2;
```

This principle works in functional languages, not so well in imperative languages.
OCAML BASICS:
TYPE CHECKING AGAIN
There are simple rules that tell you what the type of an expression is.

Those rules compute a type for an expression based on the types of its subexpressions (and the types of the variables that are in scope).

You don’t have to know the details of how a subexpression is implemented to do type checking. You just need to know its type.

That’s what makes OCaml type checking modular.

We write “e : t” to say that expression e has type t.
Example Type-checking Rules

if e : int
then string_of_int e : string
Example Type-checking Rules

if e : int
then string_of_int e : string

if e1 : bool
and e2 : t and e3 : t (the same type t, for some type t)
then if e1 then e2 else e3 : t (that same type t)
Type Checking Rules

• Violating the rules:

```bash
# "hello" + 1;;
Error: This expression has type string but an expression was expected of type int
```

• The type error message tells you the type that was expected and the type that it inferred for your subexpression

• By the way, this was one of the nonsensical expressions that did not evaluate to a value

• I consider it a good thing that this expression does not type check
Type Checking Rules

• Violating the rules:

```ocaml
# "hello" + 1;;
Error: This expression has type string but an expression was expected of type int
```

• A possible fix:

```ocaml
# "hello" ^ (string_of_int 1);;
- : string = "hello1"
```

• One of the keys to becoming a good ML programmer is to understand type error messages.
Typing Simple Let Expressions

let x = e1 in e2

x granted type of e1 for use in e2

overall expression takes on the type of e2
Typing Simple Let Expressions

**let** `x = e1 in` 

- `x` granted type of `e1` for use in `e2`
- Overall expression takes on the type of `e2`

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

- `x` has type `int` for use inside the let body
- Overall expression has type `string`
OCAML BASICS: FUNCTIONS
let add_one (x:int) : int = 1 + x ;;

Defining functions
let add_one (x:int) : int = 1 + x ;;

---

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

- Nonrecursive functions:

```ocaml
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:

```ocaml
let add_one (x:int) : int = 1 + x ;;
let add_two (x:int) : int = add_one (add_one x) ;;
```

• With a local definition:

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

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

Good style: types on top-level definitions.
Types for Functions

Some functions:

```ocaml
code
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 \( f : T_1 \rightarrow T_2 \)
and an argument \( e : T_1 \)
then \( f \ e : T_2 \)

Example:

add_one : int -> int

3 + 4 : int

add_one (3 + 4) : int
• Recall the type of add:

Definition:

```ml
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 \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f \, e : T_2 \)

\[ f : T_1 \to T_2 \quad e : T_1 \]
\[ f \, e : T_2 \]

Example:

\[ \text{add} : \text{int} \to \text{int} \to \text{int} \]
\[ 3 + 4 : \text{int} \]
\[ \text{add} \, (3 + 4) : \text{???} \]

Note:
\( A \to B \to C \) is the same as \( A \to (B \to C) \)
Rule for type-checking functions

General Rule:

\[ f : T_1 \rightarrow T_2 \quad e : T_1 \]
\[ f \ e : T_2 \]

Example:

\[ \text{add} : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \]
\[ 3 + 4 : \text{int} \]
\[ \text{add} \ (3 + 4) : \]

Remember:

\[ A \rightarrow B \rightarrow C \]
\[ \text{is the same as} \]
\[ A \rightarrow (B \rightarrow C) \]
Rule for type-checking functions

General Rule:

\[ f : T_1 \rightarrow T_2 \quad e : T_1 \]
\[ fe : T_2 \]

Remember:

A -> B -> C
is the 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:

\[ f : T_1 \rightarrow T_2 \quad e : T_1 \quad fe : T_2 \]

Remember:

A \rightarrow B \rightarrow C
is the same as
A \rightarrow (B \rightarrow C)

Example:

\[ \text{add : int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ 3 + 4 : \text{int} \]

\[ \text{add (3 + 4)} : \text{int} \rightarrow \text{int} \]

\[ (\text{add (3 + 4)}) \ 7 : \text{int} \]
Rule for type-checking functions

General Rule:

\[
f : T_1 \rightarrow T_2 \quad e : T_1 \\
fe : T_2
\]

Remember:

A -> B -> C
is the same as
A -> (B -> C)

Example:

\[
\text{add} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 : \text{int} \\
\text{add} (3 + 4) : \text{int} \rightarrow \text{int} \\
\text{add} (3 + 4) 7 : \text{int}
\]
Rule for type-checking functions

Example:

```ml
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:

```ocaml
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:

   \[ A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \]

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

   \[
   \begin{align*}
   & f \ a1 : B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a1 : A) \\
   & f \ a1 \ a2 : C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a2 : B) \\
   & f \ a1 \ a2 \ a3 : D \rightarrow E \rightarrow F \quad \text{(if } a3 : C) \\
   & f \ a1 \ a2 \ a3 \ a4 \ a5 : F \quad \text{(if } a4 : D \text{ and } a5 : E)
   \end{align*}
   \]
OUR FIRST* COMPLEX DATA STRUCTURE!
THE TUPLE

* it is really our second complex data structure since functions are data structures too!
• A tuple is a fixed, finite, ordered collection of values
• Some examples with their types:

\[
\begin{align*}
(1, 2) & : \text{int} \times \text{int} \\
("hello", 7 + 3, \text{true}) & : \text{string} \times \text{int} \times \text{bool} \\
('a', ("hello", "goodbye")) & : \text{char} \times (\text{string} \times \text{string})
\end{align*}
\]
• To use a tuple, we extract its components
• General case:

\[
\text{let } (id1, id2, \ldots, idn) = e1 \text{ in } e2
\]

• An example:

\[
\text{let } (x, y) = (2, 4) \text{ in } x + x + y
\]
• To use a tuple, we extract its components
• General case:

\[
\text{let } (id_1, id_2, \ldots, id_n) = e_1 \text{ in } e_2
\]

• An example:

\[
\text{let } (x, y) = (2, 4) \text{ in } x + x + y
\]

\[
\rightarrow 2 + 2 + 4
\]
• 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

\[ e_1 : t_1 \quad e_2 : t_2 \]
\[ (e_1, e_2) : t_1 \times t_2 \]
Rules for Typing Tuples

\[ \frac{e_1 : t_1 \quad e_2 : t_2}{(e_1, e_2) : t_1 \times t_2} \]

if \( e_1 : t_1 \times t_2 \) then
\( x_1 : t_1 \) and \( x_2 : t_2 \)
inside the expression \( e_2 \)

Overall expression takes on the type of \( e_2 \)
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.
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)
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
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

Type abbreviation

\[
\text{type point} = \text{float} \times \text{float}
\]
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

(* distance (0.0,0.0) (0.0,1.0) == 1.0
* distance (0.0,0.0) (1.0,1.0) == sqrt(1.0 + 1.0)
*)

from the picture:
* distance (x1,y1) (x2,y2) == sqrt(a^2 + b^2)

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

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

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

;;
Distance between two points

```ocaml
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))
```

Notice operators on floats have a "." in them.

Compute function results.
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
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;;
MORE TUPLES
Tuples

• Here's a tuple with 2 fields:

\[(4.0, 5.0) : \text{float} \times \text{float}\]
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
• Here's a tuple with 2 fields:

  \[(4.0, 5.0) : \text{float} \ast \text{float}\]

• Here's a tuple with 3 fields:

  \[(4.0, 5, "hello") : \text{float} \ast \text{int} \ast \text{string}\]

• Here's a tuple with 4 fields:

  \[(4.0, 5, "hello", 55) : \text{float} \ast \text{int} \ast \text{string} \ast \text{int}\]
• 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** is the tuple with zero fields!

\[
() : \text{unit}
\]

• 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!

  

  ![](image)

  - the unit value is written with a 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** is the tuple with zero fields!

\[
() : \text{unit}
\]

• 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:

\[
\text{(print_string "hello world\n") : unit}
\]

• Expressions executed for their *effect* return the unit value
SUMMARY:
BASIC FUNCTIONAL PROGRAMMING
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**
  – when the **output** has type **unit**
    • use **()** to **construct**
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 suggest how to do it
5. Build new output values
   • the result type suggest 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
5. **Build** new output values
6. Clean up by identifying repeated patterns

For tuple types:

- when the **input** has type $t_1 \times t_2$
  - use `let (x, y) = ...` to deconstruct
- when the **output** has type $t_1 \times t_2$
  - use `(e1, e2)` to **construct**

We will see this paradigm repeat itself over and over
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
type point = float * float

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

;;
Slope between two points

```ml
type point = float * float

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

;;

deconstruct tuple
```
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
    ???
;;

what can we return?

avoid divide by zero
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
```

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

- Returning an optional value from an if statement:

```plaintext
e1 : bool  e2: T  e3: T
if e1 then e2 else e3 : T

None : T option
_____________________
e : T
Some e : T option
```

```plaintext
if ... then
  None : t option
else
  Some ( ... ) : t option
```
How do we use an option?

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

returns a float option
How do we use an option?

```ocaml
slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
;;
```
How do we use an option?

`slope : point -> point -> float option`

```ocaml
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?

```ocaml
slop : 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?

```ocaml
slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  | Some s ->
  | None ->
  ;;
```

The object between | and -> is called a pattern

The "Some s" pattern includes the variable s
How do we use an option?

slop : 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

```ocaml
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)
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 in between `match ... with`

We use `(p3, p4)` here instead of `((x1, y1), (x2, y2))`

```ocaml
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))
;;
```

we built a pair of pairs
Pattern-matching in function parameters

```
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?

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.
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

```ocaml
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 SHORT JAVA RANT
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

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

```ocaml
type java_pair = (int * int) option
```
In O'Caml, if a pair may be null it is a pair option:

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

And if you write code like this:

```ocaml
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:

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

And if you write code like this:

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

You get a **helpful** error message like this:

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

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

let swap_java_pair (p:java_pair) : java_pair =
match p with
| Some (x,y) -> Some (y,x)
type java_pair = (int * int) option

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

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

**OCaml to the rescue!**

..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

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

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

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

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

An easy fix!
Moreover, your pairs are probably almost never null!

Defensive programming & always checking for null is AnNOying
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."

```ocaml
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. Your bullet-proof code is much simpler than in Java.
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 ...

OCaml 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
  - programmers have fewer cases to worry about
  - entire classes of errors just go away
  - type checking and pattern analysis help prevent programmers from ever forgetting about a case
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 to describe...
Java has a paucity of types
   ─ but at least when you forget something,
   it *throws an exception* instead of silently going off the trolley!

If you forget to check for null pointer in a C program,
   ─ no type-check error at compile time
   ─ no exception at run time
   ─ it might crash right away (that would be best), or
   ─ it might permit a buffer-overflow (or similar) vulnerability
   ─ so the hackers pwn you!
Java has a paucity of types

- but at least when you forget something, it *throws an exception* instead of going off a trolley!

If you forget to check for null pointer in a C program,
- no type-check error at compile time
- no exception at run time
- it might crash right away (that would be best), or
- it might permit a buffer-overrun (or similar) vulnerability
- so the hackers pwn you!

Summary of C, C++ rant

SCORE: OCAML 1, JAVA 0, C -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

<table>
<thead>
<tr>
<th>Type</th>
<th>Construct Values</th>
<th>Number of Cases</th>
<th>Deconstruct Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>0, -1, 2, ...</td>
<td>$2^{31}-1$</td>
<td>match $i$ with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>true, false</td>
<td>2</td>
<td>match $b$ with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1 * t2</td>
<td>(2, &quot;hi&quot;)</td>
<td>(# of t1) * (# of t2)</td>
<td>let $(x,y) = ...$ in ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>match $p$ with $(x,y)$ -&gt; ...</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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>