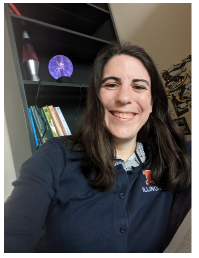
Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC



https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Quiz

Objectives for Today

- On Thursday, you learned about **environments** and **closures**, and how they track values in OCaml
  - This was motivating what actually happens when you **evaluate** an expression in OCaml
  - **We're almost there!** But we omitted a lot of important things we need to get there
  - Today, we'll cover the **remaining cool things** we need to get to evaluation
- As before, this captures concepts present in **many languages**, so it is pretty broadly useful
  - Though there are some language-specific quirks

Piazza: On optimizing closures

Questions about environments?

More about OCaml

Recall: Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three =
   fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

Again, first syntactic sugar for second

Recall: Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three =
   fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

- What is the value of add_three?
- Let ρ_{add_three} be the environment before the declaration
- Value: <x ->fun y -> (fun z -> x + y + z), ρ_{add_three} >

Recall: Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three =
   fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

- What is the value of add_three?
- Let ρ_{add_three} be the environment before the declaration
- Value: <x ->fun y -> (fun z -> x + y + z), ρ_{add_three} >

Recall: Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three =
   fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

- What is the value of add_three?
- Let $\rho_{\text{add\_three}}$ be the environment before the declaration
- Value: <x ->fun y -> (fun z -> x + y + z), $\rho_{\text{add\_three}}$ >

Partial Application

```
let add_three x y z = x + y + z
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

Partial Application

```
let add_three x y z = x + y + z
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

Partial Application

```
let add_three x y z = x + y + z

# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

Partial Application

```
let add_three x y z = x + y + z
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

Partial application also called *sectioning*

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = (fun f x -> f (f (f x))) plus_two;;

# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = (fun x ->
    plus_two (plus_two (plus_two x)));;
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let plus_six = thrice plus_two;;
val plus_six : int -> int = <fun>
# plus_six 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let plus_six = thrice plus_two;;
val plus_six : int -> int = <fun>
# plus_six 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let plus_six = thrice plus_two;;
val plus_six : int -> int = <fun>
# plus_six 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Functions as Arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let plus_six = thrice plus_two;;
val plus_six : int -> int = <fun>
# plus_six 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Questions so far?

Tuples as Values

// $\rho_1 = \{c \to 4, test \to 3.7\}$

```
# let s = (5, "hi", 3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

// $\rho_2 = \{s \to (5, \text{"hi"}, 3.2), c \to 4, test \to 3.7\}$

Tuples as Values

// $\rho_1 = \{c \rightarrow 4, test \rightarrow 3.7\}$

```
# let s = (5, "hi", 3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

// $\rho_2 = \{s \rightarrow (5, \text{"hi"}, 3.2), c \rightarrow 4, test \rightarrow 3.7\}$

Tuples as Values

// $\rho_1 = \{c \rightarrow 4, test \rightarrow 3.7\}$

```
# let s = (5, "hi", 3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

// $\rho_2 = \{s \rightarrow (5, \text{"hi"}, 3.2), c \rightarrow 4, test \rightarrow 3.7\}$

Tuples as Values

// $\rho_1 = \{c \to 4, test \to 3.7\}$

```
# let s = (5, "hi", 3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

// $\rho_2 = \{s \to (5, \text{"hi"}, 3.2), c \to 4, test \to 3.7\}$

Functions on Tuples

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3, 4);;
- : int = 7
# let double x = (x, x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Functions on Tuples

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3, 4);;
- : int = 7
# let double x = (x, x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Functions on Tuples

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3, 4);;
- : int = 7
# let double x = (x, x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Functions on Tuples

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3, 4);;
- : int = 7
# let double x = (x, x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Functions on Tuples

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3, 4);;
- : int = 7
# let double x = (x, x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Currying

Curried vs Uncurried

- Recall:

```
# let add_three u v w = u + v + w;;
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from:

```
# let add_triple (u, v, w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is *curried*;
- add_triple is *uncurried*

Curried vs Uncurried

- Recall:

```
# let add_three u v w = u + v + w;;
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from:

```
# let add_triple (u, v, w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is ***curried***;
- add_triple is *uncurried*

Curried vs Uncurried

- Recall:

```
# let add_three u v w = u + v + w;;
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from:

```
# let add_triple (u, v, w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is ***curried***;

One argument at a time

- add_triple is *uncurried*

Currying

Curried vs Uncurried

- Recall:

```
# let add_three u v w = u + v + w;;
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from:

```
# let add_triple (u, v, w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is ***curried***;
- add_triple is ***uncurried***

Curried vs Uncurried

- Recall:

```
# let add_three u v w = u + v + w;;
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from:

```
# let add_triple (u, v, w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is ***curried***;
- add_triple is ***uncurried***

Tuple, all at once

Curried vs Uncurried

```
# add_triple (6, 3, 2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'
# fun x -> add_triple (5, 4, x);;
: int -> int = <fun>
```

Curried vs Uncurried

```
# add_triple (6, 3, 2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'
# fun x -> add_triple (5, 4, x);;
: int -> int = <fun>
```

Curried vs Uncurried

```
# add_triple (6, 3, 2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'
# fun x -> add_triple (5, 4, x);;
: int -> int = <fun>
```

Questions so far?

Back to OCaml

Pattern Matching with Tuples

// ρ_1 = {s → (5, "hi", 3.2),
c → 4, a → 1, b → 5}

let **(a, b, c)** = s;; **(* (a,b,c) is a pattern *)**

val a : int = 5

val b : string = "hi"

val c : float = 3.2

Pattern Matching with Tuples

// $\rho_1 = \{s \rightarrow (5, \text{"hi"}, 3.2),$
$c \rightarrow 4, a \rightarrow 1, b \rightarrow 5\}$

```
# let (a, b, c) = s;;  (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2
```

Pattern Matching with Tuples

// ρ_1 = {s → (5, "hi", 3.2),
c → 4, a → 1, b → 5}

let **(a, b, c)** = s;; **(* (a,b,c) is a pattern *)**

val **a** : int = 5

val **b** : string = "hi"

val **c** : float = 3.2

Pattern Matching with Tuples

// ρ_1 = {s → (5, "hi", 3.2),
c → 4, a → 1, b → 5}

```
# let (a, b, c) = s;;  (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2
```

// ρ_2 = {a → 5, b →"hi", c → 3.2,
s → (5, "hi", 3.2)}

Pattern Matching with Tuples

// $\rho_1 = \{s \to (5, \text{"hi"}, 3.2),$
$c \to 4, a \to 1, b \to 5\}$

```
# let a, b, c = s;;  (* can omit parens *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2
```

// $\rho_2 = \{a \to 5, b \to \text{"hi"}, c \to 3.2,$
$s \to (5, \text{"hi"}, 3.2)\}$

Nested Tuples

```
#  (* Tuples can be nested *)
let d = ((1, 4, 62), ("bye", 15), 73.95);;
val d : (int * int * int) * (string * int) * float =
  ((1, 4, 62), ("bye", 15), 73.95)
#  (* Patterns can be nested *)
let (p, (st, _), _) = d;;
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

Nested Tuples

```
# (* Tuples can be nested *)
let d = ((1, 4, 62), ("bye", 15), 73.95);;
val d : (int * int * int) * (string * int) * float =
  ((1, 4, 62), ("bye", 15), 73.95)
# (* Patterns can be nested *)
let (p, (st, _), _) = d;;
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

Nested Tuples

```
#  (* Tuples can be nested *)
let d = ((1, 4, 62), ("bye", 15), 73.95);;
val d : (int * int * int) * (string * int) * float =
  ((1, 4, 62), ("bye", 15), 73.95)
#  (* _ matches all, but binds nothing *)
let (p, (st, _), _) = d;;
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

Closures map from *Patterns*

Last Time: Defining Closures

We lacked the vocabulary to say what this really is.

- A **closure** is a pair of:
  - an **environment**, and
  - an **association** mapping:
    - a $\boxed{\text{sequence}}$ of **variables** (input variables) to
    - an **expression** (the function body),
- written:

  $f \rightarrow < \boxed{(v1,\ldots,vn)} \rightarrow exp, \rho_f >$

- where ρ_f is the environment in effect when f is defined (if f is a simple function).

This Time: Defining Closures

- A **closure** is a pair of:
  - an **environment**, and
  - an **association** mapping:
    - a pattern of **variables** (input variables) to
    - an **expression** (the function body),
- written:

  $f \rightarrow < \boxed{(v1,...,vn)} \rightarrow exp, \rho_f >$

- where ρ_f is the environment in effect when f is defined (if f is a simple function).

Reminder: Closure for plus_x

- When plus_x was defined, we had environment:

  $\rho_{plus\_x} = \{\ldots, x \to 12, \ldots\}$

- Recall: let plus_x y = y + x

  is really let plus_x = fun y -> y + x

- Closure for fun y -> y + x:

  $<y \to y + x, \rho_{plus\_x}>$

- Environment just after plus_x defined:

  $\{plus\_x \to <y \to y + x, \rho_{plus\_x}>\} + \rho_{plus\_x}$

Reminder: Closure for plus_x

- When plus_x was defined, we had environment:

  $\rho_{plus\_x} = \{\ldots, x \rightarrow 12, \ldots\}$

- Recall: let plus_x y = y + x

  is really let plus_x = fun y -> y + x

- Closure for fun y -> y + x:

  $<\boxed{y \rightarrow}\, y + x, \rho_{plus\_x} >$

- Environment just after plus_x defined:

  $\{plus\_x \rightarrow <\boxed{y \rightarrow}\, y + x, \rho_{plus\_x} >\} + \rho_{plus\_x}$

New: Closure for plus_pair

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
```

- Assume $\rho_{\text{plus\_pair}}$ was the environment just before plus_pair defined
- Closure for fun (n,m) -> n + m:

$$<(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}}>$$

- Environment just after plus_pair defined:

$$\{ \text{plus\_pair} \rightarrow <(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} > \} + \rho_{\text{plus\_pair}}$$

New: Closure for plus_pair

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
```

- Assume $\rho_{\text{plus\_pair}}$ was the environment just before plus_pair defined
- Closure for fun (n,m) -> n + m:

$$<\boxed{(n,m)} \rightarrow n + m,\ \rho_{\text{plus\_pair}}>$$

- Environment just after plus_pair defined:

$$\{\ \text{plus\_pair} \rightarrow <\boxed{(n,m)} \rightarrow n + m,\ \rho_{\text{plus\_pair}} >\ \} + \rho_{\text{plus\_pair}}$$

New: Closure for plus_pair

```
# let plus_pair (n, m) = n + m;;
val plus_pair : int * int -> int = <fun>
```

- Assume $\rho_{\text{plus\_pair}}$ was the environment just before plus_pair defined

- Closure for fun (n,m) -> n + m:

$$< \boxed{(n,m)} \to n + m, \rho_{\text{plus\_pair}} >$$

- Environment just after plus_pair defined:

$$\{ \text{plus\_pair} \to < \boxed{(n,m)} \to n + m, \rho_{\text{plus\_pair}} > \} + \rho_{\text{plus\_pair}}$$

Questions so far?

Pattern Matching

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

val triple_to_pair : **int * int * int** -> int * int = <fun>

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

val triple_to_pair : **int * int * int** -> **int * int** = <fun>

Match Expressions

```
# let triple_to_pair triple =
  match triple with
   | (0, x, y) -> (x, y)
   | (x, 0, y) -> (x, y)
   | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
   | (0, x, y) -> (x, y)
   | (x, 0, y) -> (x, y)
   | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of only its clause

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
```

val triple_to_pair : **int * int * int** -> **int * int** = <fun>

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of **only its clause**

Use first matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of **only its clause**

Use **first** matching clause

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of **only its clause**

Use **first** matching clause

val triple_to_pair : **int * int * int** -> **int * int** = <fun>

```
# triple_to_pair (0, 5, 0);;
```

What is the result?

Match Expressions

```
# let triple_to_pair triple =
  match triple with
  | (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
```

Each clause: **pattern** on left, **expression** on right

Each x, y has scope of **only its clause**

Use **first** matching clause

val triple_to_pair : **int * int * int** -> **int * int** = <fun>

```
# triple_to_pair (0, 5, 0);;
- : int * int = (5, 0)
```

Questions?

Takeaways

- We saw some great **language features**, like:
  - tuples,
  - patterns,
  - pattern matching, and
  - partial application.
- **Currying** gets us between a function that takes a tuple as an argument, and a function that takes its arguments one at a time. The latter can be partially applied; the former cannot be!
- Closures map from **patterns**.

Next Class:
Evaluating expressions in OCaml (but actually), and more

Reminder: Also Next Class

- **WA1** is due on **Thursday**
  - This is worth points!
  - Please do this!
- **MP2** due next **Tuesday**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help

Next Class