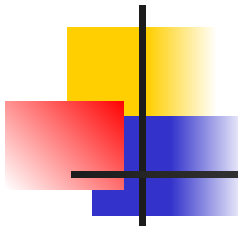


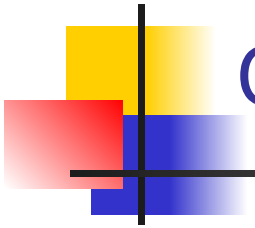
Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

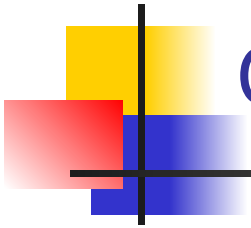
https://courses.grainger.illinois.edu/cs421/fa2023/

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



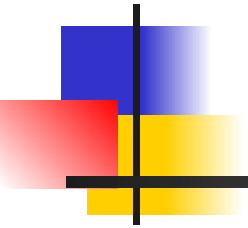
Objectives for Today

- On Tuesday, we saw some great **language features**, like tuples, patterns, pattern matching, and partial function application.
- We also saw how **currying** gets us between a function that takes a tuple as an argument, and a function that takes its arguments one at a time.
- Finally, we saw how closures map from **patterns**.
- Today, we will briefly look at **recursion** coupled with **pattern matching** in OCaml.
- We will then finally get to **evaluating expressions** in OCaml!

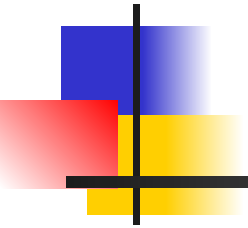


Objectives for Today

- On Tuesday, we saw some great **language features**, like tuples, patterns, pattern matching, and partial function application.
- We also saw how **currying** gets us between a function that takes a tuple as an argument, and a function that takes its arguments one at a time.
- Finally, we saw how closures map from **patterns**.
- Today, we will briefly look at **recursion** coupled with **pattern matching** in OCaml.
- We will then finally get to **evaluating expressions** in OCaml!



Questions from last time?



Intro to Recursion in OCaml

Recursive Functions

```
# (* rec needed for recursive function declarations *)
  let rec factorial n =
    if n = 0 then
      1
    else
      n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive Functions

```
# (* rec needed for recursive function declarations *)
  let rec factorial n =
    if n = 0 then
      1
    else
      n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive Functions

```
# (* rec needed for recursive function declarations *)
  let rec factorial n =
    if n = 0 then
      1
    else
      n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive Functions

```
# (* rec needed for recursive function declarations *)
  let rec factorial n =
    if n = 0 then
      1
    else
      n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive Functions

```
# (* rec needed for recursive function declarations *)
  let rec factorial n =
    if n = 0 then
      1
    else
      n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Without the **rec** keyword, we'd get "Error: Unbound value factorial."

Pattern Matching and Recursion

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec sq n = (* rec for recursion *)
   match n with  (* pattern matching for cases *)
   | 0 -> 0          (* base case *)
   | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec sq n = (* rec for recursion *)
   match n with  (* pattern matching for cases *)
   | 0 -> 0        (* base case *)
   | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec sq n = (* rec for recursion *)
   match n with (* pattern matching for cases *)
   | 0 -> 0        (* base case *)
   | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec sq n = (* rec for recursion *)
   match n with (* pattern matching for cases *)
   | 0 -> 0          (* base case *)
   | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec sq n = (* rec for recursion *)
   match n with (* pattern matching for cases *)
   | 0 -> 0          (* base case *)
   | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec sq n = (* rec for recursion *)
   match n with (* pattern matching for cases *)
   | 0 -> 0          (* base case *)
   | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

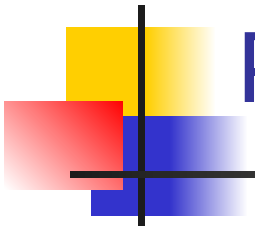
Aside: Recursion and induction are deeply, beautifully related. (Discussed in thesis, if curious.)

```
# let rec sq n = (* rec for recursion *)
    match n with (* pattern matching for cases *)
    | 0 -> 0          (* base case *)
    | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

Aside: Recursion and induction are deeply, beautifully related. (Discussed in thesis, if curious.)

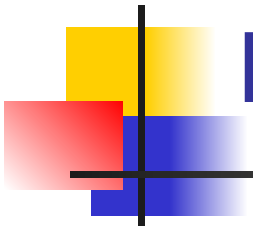
```
# let rec sq n = (* rec for recursion *)
    match n with (* pattern matching for cases *)
    | 0 -> 0          (* base case *)
    | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```



Pattern Matching and Recursion

Aside: Recursion and induction are deeply, beautifully related. (Discussed in thesis, if curious.)

```
# let rec sq n = (* rec for recursion *)
    match n with (* pattern matching for cases *)
    | 0 -> 0          (* base case *)
    | n -> (2 * n -1) + sq (n -1);; (* recursive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```



Pattern Matching and Recursion

Aside: Recursion and induction are deeply, beautifully related. (Discussed in thesis, if curious.)

```
# let rec sq n = (* rec for recursion *)
    match n with (* pattern matching for cases *)
    | 0 -> 0          (* base case *)
    | n -> (2 * n -1) + sq (n -1);; (* inductive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

Pattern Matching and Recursion

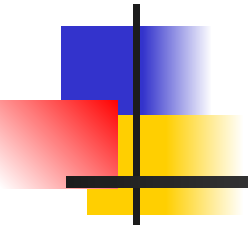
Aside: Recursion and induction are deeply, beautifully related. (Discussed in thesis, if curious.)

```
# let rec sq n = (* rec for recursion *)
    match n with (* pattern matching for cases *)
    | 0 -> 0          (* base case *)
    | n -> (2 * n -1) + sq (n -1);; (* inductive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```

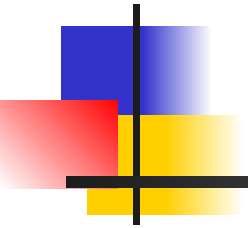
Pattern Matching and Recursion

Aside: Recursion and induction are deeply, beautifully related. (Discussed in thesis, if curious.)

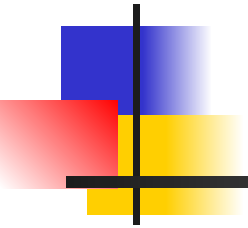
```
# let rec sq n = (* rec for recursion *)
    match n with (* pattern matching for cases *)
    | 0 -> 0          (* base case *)
    | n -> (2 * n -1) + sq (n -1);; (* inductive case *)
val sq : int -> int = <fun>
# sq 3;;
- : int = 9
```



Questions so far?



More Recursion Next Week



Evaluating OCaml Programs

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration let x = e
  - Evaluate expression e in ρ to value v
  - Update ρ with x v: $\{x \rightarrow v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not rebound in ρ_1

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, b \rightarrow 6\} =$

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration let x = e
  - Evaluate expression e in ρ to value v
  - Update ρ with x v: $\{x \rightarrow v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not rebound in ρ_1

$\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, b \rightarrow 6\} =$

$\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration let x = e
  - Evaluate expression e in ρ to value v
  - Update ρ with x v: $\{x \rightarrow v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not rebound in ρ_1

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, b \rightarrow 6\} =$

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration let x = e
  - Evaluate expression e in ρ to value v
  - Update ρ with x v: $\{x \rightarrow v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not **rebound** in ρ_1

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{\mathbf{y \rightarrow 100}, b \rightarrow 6\} =$

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration let x = e
  - Evaluate expression e in ρ to value v
  - Update ρ with x v: $\{x \rightarrow v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not **rebound** in ρ_1

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{\mathbf{y} \rightarrow \mathbf{100}, b \rightarrow 6\} =$

  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

Evaluating expressions in OCaml

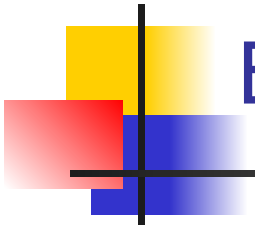
- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- A **constant** c evaluates to **itself**, including primitive operators like + and =
  - Eval (c , ρ) => Val c
- To evaluate a **variable** v, **look it up** in ρ:
  - Eval (v, ρ) => Val (ρ(v))

Evaluating expressions in OCaml

- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- A **constant** c evaluates to **itself**, including primitive operators like + and =
  - Eval (c , ρ) => Val c
- To evaluate a **variable** v, **look it up** in ρ:
  - Eval (v, ρ) => Val (ρ(v))

Evaluating expressions in OCaml

- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- A **constant** c evaluates to **itself**, including primitive operators like + and =
  - Eval (c , ρ) => Val c
- To evaluate a **variable** v, **look it up** in ρ:
  - Eval (v, ρ) => Val (ρ(v))

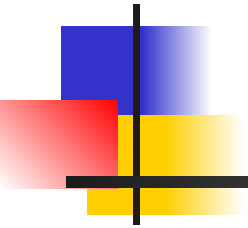


Evaluating expressions in OCaml

- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- A **constant** c evaluates to **itself**, including primitive operators like + and =
  - Eval (c , ρ) => Val c
- To evaluate a **variable** v, **look it up** in ρ:
  - Eval (v, ρ) => Val (ρ(v))

Evaluating expressions in OCaml

- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- A **constant** c evaluates to **itself**, including primitive operators like + and =
  - Eval (c , ρ) => Val c
- To evaluate a **variable** v, **look it up** in ρ:
  - Eval (v, ρ) => Val (ρ(v))



Questions so far?

Evaluating expressions in OCaml

- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- We define evaluation one *small step* at a time. So when evaluating an expression e requires recursively evaluating some subexpression, we may write something like:
  - Eval (e , ρ) => Eval (e' , ρ')

  for subexpression e' and updated environment ρ'.

Evaluating expressions in OCaml

- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- We define evaluation one *small step* at a time. So when evaluating an expression e requires recursively evaluating some subexpression, we may write something like:
  - Eval (e , ρ) => Eval (e′ , ρ′)

  for subexpression e′ and updated environment ρ′

Evaluating expressions in OCaml

- Evaluation takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- We define evaluation one *small step* at a time. So when evaluating an expression e requires recursively evaluating some subexpression, we may write something like:
  - Eval (e , ρ) => Eval (e′ , ρ′)

  for subexpression e′ and updated environment ρ′

This gives us an **algorithm** to implement an **interpreter**!

Evaluating expressions in OCaml

- To evaluate a **tuple** $(e_1 , \ldots , e_n)$:
  - Evaluate each e_i to v_i, right to left for OCaml
  - Then make value $(v_1 , \ldots , v_n)$
  - Eval$((e_1 , \ldots , e_n), \rho)$ =>
    Eval$((e_1 , \ldots , $ Eval $(e_n, \rho)), \rho)$
  - Eval$((e_1 , \ldots , e_i,$ Val $v_{i+1} , \ldots ,$ Val $v_n), \rho)$ =>
    Eval$((e_1 , \ldots ,$ Eval$(e_i, \rho),$ Val $v_{i+1} , \ldots ,$ Val $v_n), \rho)$
  - Eval$((\text{Val } v_1 , \ldots , \text{Val } v_n) , \rho)$ =>
    Val $(v_1 , \ldots , v_n)$

Evaluating expressions in OCaml

- To evaluate a **tuple** $(e_1 , \ldots , e_n)$:
  - Evaluate each e_i to v_i, **right to left** for OCaml
  - Then make value $(v_1 , \ldots , v_n)$
  - Eval$((e_1 , \ldots , e_n), \rho)$ =>
    Eval$((e_1 , \ldots , $ Eval $(e_n, \rho)), \rho)$
  - Eval$((e_1 , \ldots , e_i,$ Val $v_{i+1} , \ldots ,$ Val $v_n), \rho)$ =>
    Eval$((e_1 , \ldots ,$ Eval$(e_i, \rho),$ Val $v_{i+1} , \ldots ,$ Val $v_n), \rho)$
  - Eval$((\text{Val } v_1 , \ldots , \text{Val } v_n) , \rho)$ =>
    Val $(v_1 , \ldots , v_n)$

Evaluating expressions in OCaml

- To evaluate a **tuple** $(e_1 , \ldots , e_n)$:
  - Evaluate each e_i to v_i, **right to left** for OCaml
  - Then make **value** $(v_1 , \ldots , v_n)$
  - Eval$((e_1 , \ldots , e_n), \rho)$ =>
    Eval$((e_1 , \ldots , $ Eval $(e_n, \rho)), \rho)$
  - Eval$((e_1 , \ldots , e_i,$ Val $v_{i+1} , \ldots ,$ Val $v_n), \rho)$ =>
    Eval$((e_1 , \ldots ,$ Eval$(e_i, \rho),$ Val $v_{i+1} , \ldots ,$ Val $v_n), \rho)$
  - Eval$((\text{Val } v_1 , \ldots , \text{Val } v_n) , \rho)$ =>
    Val $(v_1 , \ldots , v_n)$

Evaluation

Evaluating expressions in OCaml

- To evaluate a **tuple** $(e_1 , \ldots , e_n)$:
  - Evaluate each e_i to v_i, **right to left** for OCaml
  - Then make **value** $(v_1 , \ldots , v_n)$
  - $Eval((e_1 , \ldots , \mathbf{e_n}), \rho) =>$
    $Eval((e_1 , \ldots , \mathbf{Eval\ (e_n, \rho)}), \rho)$
  - $Eval((e_1 , \ldots , e_i, Val\ v_{i+1} , \ldots , Val\ v_n), \rho) =>$
    $Eval((e_1 , \ldots , Eval(e_i, \rho), Val\ v_{i+1} , \ldots , Val\ v_n), \rho)$
  - $Eval((Val\ v_1 , \ldots , Val\ v_n) , \rho) =>$
    $Val\ (v_1 , \ldots , v_n)$

Evaluating expressions in OCaml

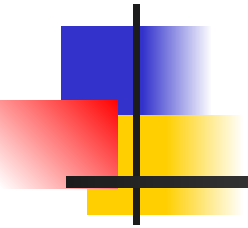
- To evaluate a **tuple** $(e_1 , \ldots , e_n)$:
  - Evaluate each e_i to v_i, **right to left** for OCaml
  - Then make **value** $(v_1 , \ldots , v_n)$
  - Eval$((e_1 , \ldots , \mathbf{e_n}), \rho)$ =>
    Eval$((e_1 , \ldots , \mathbf{Eval\ (e_n, \rho)}), \rho)$
  - Eval$((e_1 , \ldots , \mathbf{e_i}, \text{Val } v_{i+1} , \ldots , \text{Val } v_n), \rho)$ =>
    Eval$((e_1 , \ldots , \mathbf{Eval(e_i, \rho)}, \text{Val } v_{i+1} , \ldots , \text{Val } v_n), \rho)$
  - Eval$((\text{Val } v_1 , \ldots , \text{Val } v_n) , \rho)$ =>
    Val $(v_1 , \ldots , v_n)$

Evaluation

Evaluating expressions in OCaml

- To evaluate a **tuple** $(e_1 , \ldots , e_n)$:
  - Evaluate each e_i to v_i, **right to left** for OCaml
  - Then make **value** $(v_1 , \ldots , v_n)$
  - Eval$((e_1 , \ldots , \mathbf{e_n}), \rho)$ =>
    Eval$((e_1 , \ldots , \mathbf{Eval\ (e_n, \rho)}), \rho)$
  - Eval$((e_1 , \ldots , \mathbf{e_i}, \text{Val } v_{i+1} , \ldots , \text{Val } v_n), \rho)$ =>
    Eval$((e_1 , \ldots , \mathbf{Eval(e_i, \rho)}, \text{Val } v_{i+1} , \ldots , \text{Val } v_n), \rho)$
  - Eval$((\text{Val } v_1 , \ldots , \text{Val } v_n) , \rho)$ =>
    Val $(v_1 , \ldots , v_n)$

Evaluation



Questions so far?

Evaluating expressions in OCaml

- To evaluate *uses* of +, - , etc., eval **args**, then do **operation** ⊙ (+, -, *, +., …):
  - Eval(e_1 ⊙ e_2, ρ) =>
    Eval(e_1 ⊙ Eval(e_2, ρ), ρ))
  - Eval(e_1 ⊙ Val e_2, ρ) =>
    Eval(Eval(e_1, ρ) ⊙ Val v_2, ρ))
  - Eval(Val v_1 ⊙ Val v_2) =>
    Val (v_1 ⊙ v_2)
- Function expression evaluates to its closure
  - Eval (fun x -> e, ρ) => Val < x -> e, ρ>

Evaluating expressions in OCaml

- To evaluate *uses* of +, - , etc., eval **args**, then do **operation** ⊙ (+, -, *, +., …):
  - Eval(e_1 ⊙ **e_2**, ρ) =>
    Eval(e_1 ⊙ **Eval(e_2, ρ)**, ρ))
  - Eval(e_1 ⊙ Val e_2, ρ) =>
    Eval(Eval(e_1, ρ) ⊙ Val v_2, ρ))
  - Eval(Val v_1 ⊙ Val v_2) =>
    Val (v_1 ⊙ v_2)
- Function expression evaluates to its closure
  - Eval (fun x -> e, ρ) => Val < x -> e, ρ>

Evaluating expressions in OCaml

- To evaluate *uses* of +, - , etc., eval **args**, then do **operation** ⊙ (+, -, *, +., …):
  - Eval(e_1 ⊙ **e_2**, ρ) =>
    Eval(e_1 ⊙ **Eval(e_2, ρ)**, ρ))
  - Eval(**e_1** ⊙ Val e_2, ρ) =>
    Eval(**Eval(e_1, ρ)** ⊙ Val v_2, ρ))
  - Eval(Val v_1 ⊙ Val v_2) =>
    Val (v_1 ⊙ v_2)
- Function expression evaluates to its closure
  - Eval (fun x -> e, ρ) => Val < x -> e, ρ>

Evaluating expressions in OCaml

- To evaluate *uses* of +, - , etc., eval **args**, then do **operation** ⊙ (+, -, *, +., …):
  - Eval(e_1 ⊙ **e_2**, ρ) =>
    Eval(e_1 ⊙ **Eval(e_2, ρ)**, ρ))
  - Eval(**e_1** ⊙ Val e_2, ρ) =>
    Eval(**Eval(e_1, ρ)** ⊙ Val v_2, ρ))
  - Eval(Val v_1 ⊙ Val v_2, ρ) =>
    Val (v_1 ⊙ v_2)
- Function expression evaluates to its closure
  - Eval (fun x -> e, ρ) => Val < x -> e, ρ>

Evaluating expressions in OCaml

- To evaluate *uses* of +, - , etc., eval **args**, then do **operation** ⊙ (+, -, *, +., …):
  - Eval(e_1 ⊙ **e_2**, ρ) =>
    Eval(e_1 ⊙ **Eval(e_2, ρ)**, ρ))
  - Eval(**e_1** ⊙ Val e_2, ρ) =>
    Eval(**Eval(e_1, ρ)** ⊙ Val v_2, ρ))
  - Eval(Val v_1 ⊙ Val v_2) =>
    Val (v_1 ⊙ v_2)
- **Function** expression evaluates to its **closure**
  - Eval (fun x -> e, ρ) => Val < x -> e, ρ>

Evaluating expressions in OCaml

- To evaluate a **local declaration**:
  let x = e1 in e2
  - Eval e1 to v, then eval e2 using $\{x \rightarrow v\} + \rho$
  - Eval(let x = $\mathbf{e_1}$ in e_2, ρ) =>
    Eval(let x = **Eval(e_1, ρ)** in e_2, ρ)
  - Eval(let x = Val v in e_2, ρ) =>
    Eval(e_2, $\{x \rightarrow v\} + \rho$)

Evaluating expressions in OCaml

- To evaluate a **local declaration**:
  let x = e1 in e2
  - Eval e1 to v, then eval e2 using $\{x \rightarrow v\} + \rho$
  - Eval(let x = $\mathbf{e_1}$ in e_2, ρ) =>
    Eval(let x = **Eval($\mathbf{e_1}$, ρ)** in e_2, ρ)
  - Eval(let x = Val v in e_2, ρ) =>
    Eval(e_2, $\{x \rightarrow v\} + \rho$)

Evaluating expressions in OCaml

- To evaluate a **local declaration**:
  let x = e1 in e2
  - Eval e1 to v, then eval e2 using $\{x \rightarrow v\} + \rho$
  - Eval(let x = $\mathbf{e_1}$ in e_2, ρ) =>
    Eval(let x = **Eval($\mathbf{e_1}$, ρ)** in e_2, ρ)
  - Eval(let x = Val v in $\mathbf{e_2}$, ρ) =>
    Eval($\mathbf{e_2}$, $\{x \rightarrow v\} + \rho$)

Evaluating expressions in OCaml

- To evaluate a **conditional** expression:

  if b then e_1 else e_2

  - Evaluate b to a value v
  - If v is true, evaluate e_1
  - If v is false, evaluate e_2
  - Eval(if **b** then e_1 else e_2, ρ) =>
    Eval(if **Eval(b, ρ)** then e_1 else e_2, ρ)
  - Eval(if Val **true** then e_1 else e_2, ρ) => Eval(**e_1**, ρ)
  - Eval(if Val **false** then e_1 else e_2, ρ) => Eval(**e_2**, ρ)

Evaluating expressions in OCaml

- To evaluate a **conditional** expression:

  if b then e_1 else e_2

  - Evaluate b to a value v
  - If v is true, evaluate e_1
  - If v is false, evaluate e_2
  - Eval(if **b** then e_1 else e_2, ρ) =>
    Eval(if **Eval(b, ρ)** then e_1 else e_2, ρ)
  - Eval(if Val **true** then e_1 else e_2, ρ) => Eval(**e_1**, ρ)
  - Eval(if Val **false** then e_1 else e_2, ρ) => Eval(**e_2**, ρ)

Evaluating expressions in OCaml

- To evaluate a **conditional** expression:

  if b then e_1 else e_2

  - Evaluate b to a value v
  - If v is true, evaluate e_1
  - If v is false, evaluate e_2
  - Eval(if **b** then e_1 else e_2, ρ) =>
    Eval(if **Eval(b, ρ)** then e_1 else e_2, ρ)
  - Eval(if Val **true** then e_1 else e_2, ρ) => Eval($\mathbf{e_1}$, ρ)
  - Eval(if Val **false** then e_1 else e_2, ρ) => Eval($\mathbf{e_2}$, ρ)

Evaluating expressions in OCaml

- To evaluate a **conditional** expression:

  if b then e_1 else e_2

  - Evaluate b to a value v
  - If v is true, evaluate e_1
  - If v is false, evaluate e_2
  - Eval(if **b** then e_1 else e_2, ρ) =>
    Eval(if **Eval(b, ρ)** then e_1 else e_2, ρ)
  - Eval(if Val **true** then e_1 else e_2, ρ) => Eval(**e_1**, ρ)
  - Eval(if Val **false** then e_1 else e_2, ρ) => Eval(**e_2**, ρ)

Evaluating expressions in OCaml

- To evaluate a **conditional** expression:

  if b then e_1 else e_2

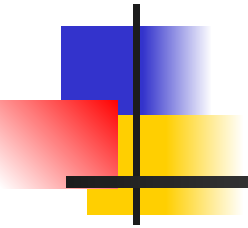
  - Evaluate b to a value v
  - If v is true, evaluate e_1
  - If v is false, evaluate e_2
  - Eval(if **b** then e_1 else e_2, ρ) =>
    Eval(if **Eval(b, ρ)** then e_1 else e_2, ρ)
  - Eval(if Val **true** then e_1 else e_2, ρ) => Eval(**e_1**, ρ)
  - Eval(if Val **false** then e_1 else e_2, ρ) => Eval(**e_2**, ρ)

Evaluating expressions in OCaml

- To evaluate a **conditional** expression:

  if b then e_1 else e_2

  - Evaluate b to a value v
  - If v is true, evaluate e_1
  - If v is false, evaluate e_2
  - Eval(if **b** then e_1 else e_2, ρ) =>
    Eval(if **Eval(b, ρ)** then e_1 else e_2, ρ)
  - Eval(if Val **true** then e_1 else e_2, ρ) => Eval($\mathbf{e_1}$, ρ)
  - Eval(if Val **false** then e_1 else e_2, ρ) => Eval($\mathbf{e_2}$, ρ)



Questions so far?

Evaluation of Application with Closures

- Given **application** expression f e
  - In OCaml, evaluate e to value v
  - In environment ρ, evaluate f to $<(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - Evaluate body b in $\{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
  - Eval(f **e**, ρ) => Eval(f (**Eval(e, ρ)**), ρ)
  - Eval(**f** (Val v), ρ) => Eval((**Eval(f, ρ)**) (Val v), ρ)
  - $\text{Eval}((\text{Val}<(x_1 , \ldots , x_n)\rightarrow b, \rho'>)(\text{Val}(v_1 , \ldots , v_n)), \rho)$ => $\text{Eval}(b, \{x_1\rightarrow v_1 , \ldots , x_n\rightarrow v_n\} + \rho')$

Evaluation of Application with Closures

- Given **application** expression f e
  - In OCaml, evaluate e to value v
  - In environment ρ, evaluate f to $<(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - Evaluate body b in $\{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
  - Eval(f **e**, ρ) => Eval(f (**Eval(e, ρ)**), ρ)
  - Eval(**f** (Val v), ρ) => Eval((**Eval(f, ρ)**) (Val v), ρ)
  - $\text{Eval}((\text{Val}<(x_1 , \ldots , x_n)\rightarrow b, \rho'>)(\text{Val}(v_1 , \ldots , v_n)), \rho)$ $\Rightarrow \text{Eval}(b, \{x_1\rightarrow v_1 , \ldots , x_n\rightarrow v_n\} + \rho')$

Evaluation of Application with Closures

- Given **application** expression f e
  - In OCaml, evaluate e to value v
  - In environment ρ, evaluate f to $<(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - Evaluate body b in $\{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
  - Eval(f **e**, ρ) => Eval(f (**Eval(e, ρ)**), ρ)
  - Eval(**f** (Val v), ρ) => Eval((**Eval(f, ρ)**) (Val v), ρ)
  - $\text{Eval}((\text{Val}<(x_1 , \ldots , x_n)\rightarrow b, \rho'>)(\text{Val}(v_1 , \ldots , v_n)), \rho)$ => $\text{Eval}(b, \{x_1\rightarrow v_1 , \ldots , x_n\rightarrow v_n\} + \rho')$

Evaluation of Application with Closures

- Given **application** expression f e
  - In OCaml, evaluate e to value v
  - In environment ρ, evaluate f to $<(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - Evaluate body b in $\{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
  - Eval(f **e**, ρ) => Eval(f (**Eval(e, ρ)**), ρ)
  - Eval(f **(Val v)**, ρ) => Eval((**Eval(f, ρ)**) (Val v), ρ)
  - $Eval((Val<(x_1 , \ldots , x_n)\rightarrow b, \rho'>)(Val(v_1 , \ldots , v_n)), \rho)$ => $Eval(b, \{x_1\rightarrow v_1 , \ldots , x_n\rightarrow v_n\} + \rho')$

Evaluation of Application with Closures

- Given **application** expression f e
  - In OCaml, evaluate e to value v
  - In environment ρ, evaluate f to $<(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - Evaluate body b in $\{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
  - Eval(f **e**, ρ) => Eval(f (**Eval(e, ρ)**), ρ)
  - Eval(**f** (Val v), ρ) => Eval((**Eval(f, ρ)**) (Val v), ρ)
  - $\text{Eval}((\text{Val}<(x_1 , \ldots , x_n)\rightarrow b, \rho'>)(\text{Val}(v_1 , \ldots , v_n)), \rho)$ => $\text{Eval}(b, \{x_1\rightarrow v_1 , \ldots , x_n\rightarrow v_n\} + \rho')$

Evaluation of Application with Closures

- Given **application** expression f e
  - In OCaml, evaluate e to value v
  - In environment ρ, evaluate f to $<(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - Evaluate body b in $\{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
  - Eval(f **e**, ρ) => Eval(f (**Eval(e, ρ)**), ρ)
  - Eval(**f** (Val v), ρ) => Eval((**Eval(f, ρ)**) (Val v), ρ)
  - $\text{Eval}((\text{Val}<(x_1 , \ldots , x_n)\rightarrow b, \rho'>)(\text{Val}(v_1 , \ldots , v_n)), \rho)$ => $\text{Eval}(b, \{x_1\rightarrow v_1 , \ldots , x_n\rightarrow v_n\} + \rho')$

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x **z**, ρ) =>

  Your turn!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x **z**, ρ) =>

  Eval (plus_x **(Eval(z, ρ))**) =>

  Keep going!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x **z**, ρ) =>

  Eval (plus_x **(Eval(z, ρ))**) =>

  Eval (plus_x **(Val ?)**, ρ) =>

  Look it up!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x **z**, ρ) =>

  Eval (plus_x **(Eval(z, ρ))**) =>

  Eval (plus_x **(Val ?)**, ρ) =>

  Look it up!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x **z**, ρ) =>

  Eval (plus_x **(Eval(z, ρ))**) =>

  Eval (plus_x **(Val 3)**, ρ) =>

  Done with argument!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x z, ρ) =>

  Eval (plus_x (Eval(z, ρ))) =>

  Eval (**plus_x** (Val 3), ρ) =>

  Now what?

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x z, ρ) =>

  Eval (plus_x (Eval(z, ρ))) =>

  Eval (**plus_x** (Val 3), ρ) =>

  Eval (**(Eval (plus_x, ρ))** (Val 3), ρ) =>

  Now what?

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x z, ρ) =>

  Eval (plus_x (Eval(z, ρ))) =>

  Eval (**plus_x** (Val 3), ρ) =>

  Eval (**(Eval (plus_x, ρ))** (Val 3), ρ) =>

  Eval (**(Val ?)**(Val 3), ρ) =>

  Look it up!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x z, ρ) =>

  Eval (plus_x (Eval(z, ρ))) =>

  Eval (**plus_x** (Val 3), ρ) =>

  Eval (**(Eval (plus_x, ρ))** (Val 3), ρ) =>

  Eval (**(Val ?)**(Val 3), ρ) =>

  Look it up!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$ 17, z $\rightarrow$ 3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x z, ρ) =>

  Eval (plus_x (Eval(z, ρ))) =>

  Eval (**plus_x** (Val 3), ρ) =>

  Eval (**(Eval (plus_x, ρ))** (Val 3), ρ) =>

  Eval (**(Val <y $\rightarrow$ y + x, ρ_{plus_x} >)**(Val 3), ρ) =>

  Closure!

Evaluation

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (plus_x z, ρ) =>

  Eval (plus_x (Eval(z, ρ))) =>

  Eval (plus_x (Val 3), ρ) =>

  Eval ((Eval (plus_x, ρ)) (Val 3), ρ) =>

  Eval (**(Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3)**, ρ) =>

  What's next?

Evaluation

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (**(Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3)**, ρ) =>

  How to evaluate closure?

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval (**(Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3)**, ρ) =>

  Eval (**y + x**, …) =>

  Evaluate its body …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((**Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3)**, ρ) =>

  Eval (y + x, **? + ρ_{plus_x}**) =>

Evaluate its body in an updated environment!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((**Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3)**, ρ) =>

  Eval (y + x, **{y $\rightarrow$ 3} + ρ_{plus_x}**) =>

Evaluate its body in an updated environment!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (**y + x**, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  How to evaluate applied operator?

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + **x**, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + **Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x})**, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  RHS first!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + **Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}**), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Look it up!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + **Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}**), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + **Val 12**), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Now what?

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**y** + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  LHS next!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**y** + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**Eval(y, {y $\rightarrow$ 3} + ρ_{plus_x})** + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  LHS next!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**Eval(y, {y $\rightarrow$ 3} + ρ_{plus_x})** + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Look it up in which environment?

Evaluation

*

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**Eval(y, {y $\rightarrow$ 3} + ρ_{plus_x})** + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Look it up locally!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**Eval(y, {y $\rightarrow$ 3} + ρ_{plus_x})** + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**Val 3** + Val 12 , {y $\rightarrow$ 3} + ρ_{plus_x}) =>

Look it up locally!

Evaluation

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (Eval(y, {y $\rightarrow$ 3} + ρ_{plus_x}) + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**Val 3 + Val 12**, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

Finally, the operator!

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (Eval(y, {y $\rightarrow$ 3} + ρ_{plus_x}) + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (**Val 3 + Val 12**, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Val (3 + 12) = Val 15

Evaluation of Application of plus_x;;

- Have environment:

  ρ = { plus_x $\rightarrow$<y $\rightarrow$ y + x, ρ_{plus_x} >, … ,
  y $\rightarrow$ 19, x $\rightarrow$17, z $\rightarrow$3, … }

  where: ρ_{plus_x} = {x $\rightarrow$ 12, … , y $\rightarrow$ 24, …}

- Eval ((Val <y $\rightarrow$ y + x, ρ_{plus_x} >)(Val 3), ρ) =>

  Eval (y + x, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (y + Eval(x, {y $\rightarrow$ 3} + ρ_{plus_x}), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

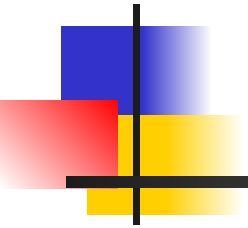
  Eval (y + Val 12), {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (Eval(y, {y $\rightarrow$ 3} + ρ_{plus_x}) + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Eval (Val 3 + Val 12, {y $\rightarrow$ 3} + ρ_{plus_x}) =>

  Val (3 + 12) = **Val 15**

Evaluation



Questions so far?

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , \text{plus\_pair} \rightarrow \langle \mathbf{(n, m)} \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$

$+ \rho_{\text{plus\_pair}}$

- Eval (plus_pair **(4, x)**, ρ) =>

Eval (plus_pair (Eval ((4, x), ρ)), ρ) =>

Eval (plus_pair (Eval ((4, Eval (x , ρ)), ρ)), ρ) =>

Eval (plus_pair (Eval ((4, Val 3), ρ)), ρ) =>

Eval (plus_pair (Eval ((Eval (4, ρ), Val 3), ρ)), ρ) =>

Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , \text{plus\_pair} \rightarrow \langle \mathbf{(n, m)} \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$

$+ \rho_{\text{plus\_pair}}$

- Eval (plus_pair **(4, x)**, ρ) =>

  Eval (plus_pair (**Eval ((4, x), ρ)**), ρ) =>

  Eval (plus_pair (Eval ((4, Eval (x , ρ)), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Eval (4, ρ), Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , \text{plus\_pair} \rightarrow \langle \mathbf{(n, m)} \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$

$+ \rho_{\text{plus\_pair}}$

- Eval (plus_pair **(4, x)**, ρ) =>

  Eval (plus_pair (Eval ((4, **x**), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, Eval (x , ρ)), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Eval (4, ρ), Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , \text{plus\_pair} \rightarrow \langle \mathbf{(n, m)} \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$

$+ \rho_{\text{plus\_pair}}$

- Eval (plus_pair (4, x), ρ) =>

  Eval (plus_pair (Eval ((4, **x**), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, **Eval (x , ρ)**), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Eval (4, ρ), Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , \text{plus\_pair} \rightarrow \langle \mathbf{(n, m)} \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$

$+ \rho_{\text{plus\_pair}}$

- Eval (plus_pair (4, x), ρ) =>

  Eval (plus_pair (Eval ((4, **x**), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, **Eval (x , ρ)**), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, **Val 3**), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Eval (4, ρ), Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , \text{plus\_pair} \rightarrow \langle \mathbf{(n, m)} \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$

$+ \rho_{\text{plus\_pair}}$

- Eval (plus_pair (4, x), ρ) =>

  Eval (plus_pair (Eval ((4, x), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, Eval (x , ρ)), ρ)), ρ) =>

  Eval (plus_pair (Eval ((**4**, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Eval (4, ρ), Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (4, x), ρ) =>

  Eval (plus_pair (Eval ((4, x), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, Eval (x , ρ)), ρ)), ρ) =>

  Eval (plus_pair (Eval ((**4**, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((**Eval (4, ρ)**, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , \text{plus\_pair} \rightarrow \langle \mathbf{(n, m)} \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$

$+ \rho_{\text{plus\_pair}}$

- Eval (plus_pair (4, x), ρ) =>

  Eval (plus_pair (Eval ((4, x), ρ)), ρ) =>

  Eval (plus_pair (Eval ((4, Eval (x , ρ)), ρ)), ρ) =>

  Eval (plus_pair (Eval ((**4**, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((**Eval (4, ρ)**, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Eval ((**Val 4**, Val 3), ρ)), ρ) =>

Evaluation

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , plus\_pair \rightarrow <\mathbf{(n, m)} \rightarrow n + m, \rho_{plus\_pair}>\}$

$+ \rho_{plus\_pair}$

- Eval (plus_pair (Eval (**(Val 4, Val 3)**, ρ)), ρ) =>

  Eval (plus_pair (Val (4, 3)), ρ) =>

  Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) => ...

  Eval ((Val<(n,m)→n+m, ρ_{plus_pair}>)(Val(4,3)) , ρ)=>

  Eval (Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Eval (Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Val 7

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (Eval (**(Val 4, Val 3)**, ρ)), ρ) =>

  Eval (plus_pair (**Val (4, 3)**), ρ) =>

  Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) => …

  Eval ((Val<(n,m)→n+m, ρ_{plus_pair}>)(Val(4,3)) , ρ)=>

  Eval (Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Eval (Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Val 7

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3 , \ldots , plus\_pair \rightarrow <\mathbf{(n, m)} \rightarrow n + m, \rho_{plus\_pair}>\}$

$+ \rho_{plus\_pair}$

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

  Eval (**plus_pair** (Val (4, 3)), ρ) =>

  Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) => ...

  Eval ((Val<(n,m)→n+m, ρ_{plus_pair}>)(Val(4,3)) , ρ)=>

  Eval (Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Eval (Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Val 7

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

  Eval (**plus_pair** (Val (4, 3)), ρ) =>

  Eval (**Eval (plus_pair, ρ)**, Val (4, 3)), ρ) =>

  Eval ((Val<(n,m)→n+m, ρ_{plus_pair}>)(Val(4,3)) , ρ)=>

  Eval (Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Eval (Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Val 7

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

  Eval (**plus_pair** (Val (4, 3)), ρ) =>

  Eval (**Eval (plus_pair, ρ)**, Val (4, 3)), ρ) =>

  Eval (**(Val<(n,m)→n+m, ρ_{plus_pair}>**)(Val (4 , 3)) , ρ)=>

  Eval (Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Eval (Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Val 7

Evaluation

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Val (4, 3)), ρ) =>

  Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) =>

  Eval (**(Val<(n,m)→n+m, ρ_{plus_pair}>)(Val (4 , 3))** , **ρ**)=>

  Eval (Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Eval (Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Val 7

Evaluation

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Val (4, 3)), ρ) =>

  Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) =>

  Eval (**(Val<(n,m)→n+m, ρ_{plus_pair}>)(Val (4 , 3)) , ρ**)=>

  Eval **(Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair})** =>

  Eval (Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair}) =>

  Val 7

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Val (4, 3)), ρ) =>

  Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) =>

  Eval (**(Val<(n,m)→n+m, ρ_{plus_pair}>)(Val (4 , 3)) , ρ**)=>

  Eval **(Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair})** =>

  Eval **(Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair})** =>

  Val 7

Evaluation

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3 , … , plus_pair →<**(n, m)** →n + m, ρ_{plus_pair}>}

+ ρ_{plus_pair}

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

  Eval (plus_pair (Val (4, 3)), ρ) =>

  Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) =>

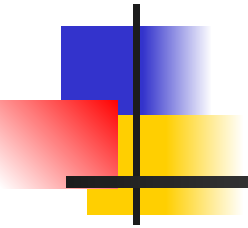
  Eval (**(Val<(n,m)→n+m, ρ_{plus_pair}>)(Val (4 , 3)) , ρ**)=>

  Eval **(Val (n + m), {n -> 4, m -> 3} + ρ_{plus_pair})** =>

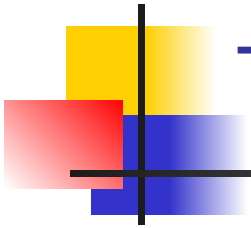
  Eval **(Val (4 + 3), {n -> 4, m -> 3} + ρ_{plus_pair})** =>

* **Val 7**

Evaluation



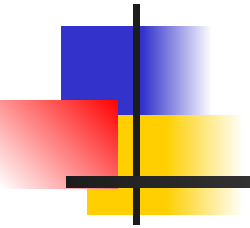
Questions?



Takeaways

- To use **recursion** in OCaml, you need the **rec** keyword
- **Evaluation** takes expression e and an environment ρ, and evaluates e in ρ to some value v:
  - Eval (e , ρ) => v
- We define evaluation **one *small step* at a time**. So when evaluating an expression e requires recursively evaluating, we write:
  - Eval (e , ρ) => Eval (e′ , ρ′)

  for subexpression e′ and updated environment ρ′
- This gives an **algorithm** for an **interpreter**!



Next Class:
Lists, more recursion

Reminder: Also Next Class

- **MP2** due **Tuesday**
  - This is worth points!
  - Please do this!
- **WA2** due next **Thursday**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help