Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2023/CS421D

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Functions with more than one argument

\# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
\# let t = add_three 6 3 2;;
val t : int = 11
\# 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

Functions with more than one argument

\# let add_three x y 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
- Remember:

let add_three =
 fun x -> (fun y -> (fun z -> x + y + z));;
Value: <x ->fun y -> (fun z -> x + y + z), ρ_{add_three} >

Partial application of functions

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 of functions

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

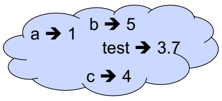
- Partial application also called *sectioning*

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!"

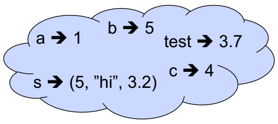
Tuples as Values

// ρ_7 = {c $\rightarrow$ 4, test $\rightarrow$ 3.7,
a $\rightarrow$ 1, b $\rightarrow$ 5}



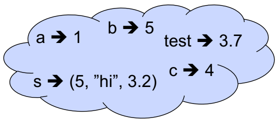

```
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

// ρ_8 = {s $\rightarrow$ (5, "hi", 3.2),
c $\rightarrow$ 4, test $\rightarrow$ 3.7,
a $\rightarrow$ 1, b $\rightarrow$ 5}

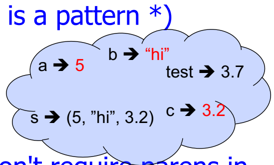


Pattern Matching with Tuples

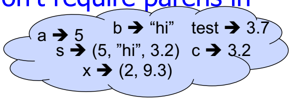
/ ρ_8 = {s $\rightarrow$ (5, "hi", 3.2),
c $\rightarrow$ 4, test $\rightarrow$ 3.7,
a $\rightarrow$ 1, b $\rightarrow$ 5}




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




```
# let x = 2, 9.3;; (* tuples don't require parens in
  Ocaml *)
val x : int * float = (2, 9.3)
```



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;; (* _ matches all, binds nothing
  *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```


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")
```


Curried vs Uncurried

- Recall

```
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

```
# 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>
```

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

Save the Environment!

- A *closure* is a pair of an environment and an association of a pattern (e.g. (v1,…,vn) giving the input variables) with an expression (the function body), written:

  $< (v1,\ldots,vn) \to exp, \rho >$

- Where ρ is the environment in effect when the function is defined (for a simple function)

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for fun (n,m) -> n + m:

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

- Environment just after plus_pair defined:

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

  $+ \rho_{plus\_pair}$

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 $\to$ v: $\{x \to v\} + \rho$

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: $\{x \to v\} + \rho$
- 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

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

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

Evaluating expressions in OCaml

- Evaluation uses an environment ρ
- A constant evaluates to itself, including primitive operators like + and =

Evaluating expressions in OCaml

- Evaluation uses an environment ρ
- A constant evaluates to itself, including primitive operators like + and =
- To evaluate a variable, look it up in ρ: $\rho(v)$

Evaluating expressions in OCaml

- Evaluation uses an environment ρ
- A constant evaluates to itself, including primitive operators like + and =
- To evaluate a variable, look it up in ρ: $\rho(v)$
- To evaluate a tuple $(e_1,\ldots,e_n)$,
  - Evaluate each e_i to v_i, right to left for Ocaml
  - Then make value $(v_1,\ldots,v_n)$

Evaluating expressions in OCaml

- To evaluate uses of +, - , etc, eval args, then do operation

Evaluating expressions in OCaml

- To evaluate uses of +, - , etc, eval args, then do operation
- Function expression evaluates to its closure

Evaluating expressions in OCaml

- To evaluate uses of +, - , etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: let x = e1 in e2
  - Eval e1 to v, then eval e2 using $\{x \to v\} + \rho$

Evaluating expressions in OCaml

- To evaluate uses of +, - , etc, eval args (right to left for Ocaml), then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: let x = e1 in e2
  - Eval e1 to v, then eval e2 using $\{x \to v\} + \rho$
- To evaluate a conditional expression: if b then e1 else e2
  - Evaluate b to a value v
  - If v is True, evaluate e1
  - If v is False, evaluate e2

Evaluation of Application with Closures

- Given application expression f e
- In Ocaml, evaluate e to value v
- In environment ρ, evaluate left term to closure, $c = <(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - $(x_1,\ldots,x_n)$ variables in (first) argument
  - v must have form $(v_1,\ldots,v_n)$
- Update the environment ρ' to
  $\rho'' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
- Evaluate body b in environment ρ''

Extra Material for Extra Credit

Evaluating expressions in OCaml

- Evaluation uses an environment ρ
  - Eval (e , ρ)
- A constant 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

- 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($(e_1,\ldots,e_n)$,ρ)=> Eval($(e_1,\ldots,$Eval $(e_n, \rho))$, ρ)
  - Eval($(e_1,\ldots,e_i$, Val $v_{i+1},\ldots,$ Val $v_n)$, ρ) => Eval($(e_1,\ldots,$Eval(e_i, ρ), Val $v_{i+1},\ldots,$ Val $v_n)$, ρ)
  - Eval((Val $v_1,\ldots,$Val v_n) , ρ) => Val $(v_1,\ldots,v_n)$

Evaluating expressions in OCaml

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

Evaluating expressions in OCaml

- To evaluate a local dec: let x = e1 in e2
  - Eval e1 to v, then eval e2 using $\{x \rightarrow v\} + \rho$
  - Eval(let x = 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 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)

Evaluation of Application with Closures

- Given application expression f e
- In Ocaml, evaluate e to value v
- In environment ρ, evaluate left term to closure,
  c = <(x_1,…,x_n) → b, ρ'>
  - (x_1,…,x_n) variables in (first) argument
  - v must have form (v_1,…,v_n)
- Update the environment ρ' to
  ρ'' = {x_1 → v_1,…, x_n →v_n}+ ρ'
- Evaluate body b in environment ρ''

Evaluation of Application with Closures

- 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,…,x_n) → b, ρ'>)(Val (v_1,…,v_n)), ρ)=>
  Eval(b, {x_1 → v_1,…, x_n → v_n}+ρ')

Evaluation of Application of plus_x;;

- Have environment:
  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}
  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval (plus_x z, ρ) =>
- Eval(plus_x (Eval(z, ρ))) => …

Evaluation of Application of plus_x;;

- Have environment:
  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}
  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval (plus_x z, ρ) =>
- Eval(plus_x (Eval(z, ρ)), ρ) =>
- Eval(plus_x (Val 3), ρ) => …

Evaluation of Application of plus_x;;

- Have environment:
  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}
  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval (plus_x z, ρ) =>
- Eval (plus_x (Eval(z, ρ)), ρ) =>
- Eval (plus_x (Val 3), ρ) =>
- Eval ((Eval(plus_x, ρ)) (Val 3), ρ) => …

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)), 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) => …

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) => …

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}) => …

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}) => …

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}) => …

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(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}) =>…

Evaluation of Application of plus_x;;

- Have environment:

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

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval(Eval(y, {y → 3} +ρ_{plus_x}) +
  Val 12,{y → 3} +ρ_{plus_x}) =>
- Eval(Val 3 + Val 12 ,{y → 3} +ρ_{plus_x}) =>…

Evaluation of Application of plus_x;;

- Have environment:

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

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval(Eval(y, {y → 3} +ρ_{plus_x}) +
  Val 12,{y → 3} +ρ_{plus_x}) =>
- Eval(Val 3 + Val 12 ,{y → 3} +ρ_{plus_x}) =>
- Val (3 + 12) = Val 15

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 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 (n + m, {n -> 4, m -> 3} + ρ_{plus_pair}) =>
- Eval (4 + 3, {n -> 4, m -> 3} + ρ_{plus_pair}) => 7

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun n -> n + 5;;
(* 0 *)
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
let a = f (4,6);;
```

What is the environment at (* 0 *)?

Answer

let f = fun n -> n + 5;;

ρ_0 = {f → <n → n + 5, { }>}

Closure question

- If we start in an empty environment, and we execute:

let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
(* 1 *)
let f = pair_map f;;
let a = f (4,6);;

What is the environment at (* 1 *)?

Answer

ρ_0 = {f → <n → n + 5, { }>}
let pair_map g (n,m) = (g n, g m);;

ρ_1 = {pair_map →
<g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>,
f → <n → n + 5, { }>}

Closure question

- If we start in an empty environment, and we execute:

let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
(* 2 *)
let a = f (4,6);;

What is the environment at (* 2 *)?

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}
ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}
let f = pair_map f;;

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}
ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}
Eval(pair_map f, ρ_1) =

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}
ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}
Eval(pair_map f, ρ_1) =>
Eval(pair_map (Eval(f, ρ_1)), ρ_1) =>
Eval(pair_map (Val<n → n + 5, { }>), ρ_1) =>
Eval((Eval(pair_map, ρ_1))(Val<n → n+5, { }>), ρ_1) =>
Eval((Val (<g→fun (n,m) -> (g n, g m), ρ_0>)
(Val <n → n + 5, { }>), ρ_1) =>
Eval(fun (n,m)->(g n, g m), {g→<n→n + 5, { }>}+ρ_0)
=>

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}
ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}
Eval(pair_map f, ρ_1) => … =>
Eval(fun (n,m)->(g n, g m), {g→<n→n + 5, { }>}+ρ_0) =
Eval(fun (n,m)->(g n, g m),
{g→<n→n + 5, { }>, f → <n → n + 5, { }>}) =>
Val (<(n,m) →(g n, g m),
{g→<n→n + 5, { }>, f → <n → n + 5, { }>})

Answer

ρ_1 = {pair_map →
<g → fun (n,m) -> (g n, g m),{f → <n → n + 5, { }>}>,
f → <n → n + 5, { }>}
let f = pair_map f;;
ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}
(*Remember: the original f is now removed from ρ_2 *)

Closure question

- If we start in an empty environment, and we execute:

let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
let a = f (4,6);;
(* 3 *)
What is the environment at (* 3 *)?

Final Evalution?

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}
let a = f (4,6);;

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}
Eval(f (4,6), ρ_2) =

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}
Eval(f (4,6), ρ_2) => Eval(f (Eval((4,6), ρ_2)), ρ_2) =>
Eval(f (Eval((4,Eval(6, ρ_2)), ρ_2)), ρ_2) =>
Eval(f (Eval((4,Val 6), ρ_2)), ρ_2) =>
Eval(f (Eval((Eval(4, ρ_2),Val 6), ρ_2)), ρ_2) =>
Eval(f (Eval((Val 4,Val 6), ρ_2)), ρ_2) =>

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}

Eval(f (4,6), ρ_2) => … =>
Eval(f (Eval((Val 4,Val 6), ρ_2)), ρ_2) =>
Eval(f (Val (4, 6)), ρ_2) =>
Eval(Eval(f, ρ_2) (Val (4, 6)), ρ_2) =>

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}

Eval(f (4,6), ρ_2) => … =>
Eval(Eval(f, ρ_2) (Val (4, 6)), ρ_2) =>
Eval((Val <(n,m)→(g n, g m),
{g→<n→n+5, { }>,
f→<n→n+5, { }>}>)(Val(4,6)))),ρ_2)=>

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}

Eval((Val <(n,m)→(g n, g m),
{g→<n→n+5, { }>,
f→<n→n+5, { }>}>)(Val(4,6)))),ρ_2)=>
Eval((g n, g m), {n → 4, m → 6, g→<n→n+5, { }>,
f→<n→n+5, { }>}) =>

Evaluate f (4,6);;

Let ρ' = {n → 4, m → 6, g→<n→n+5, { }>,
f→<n→n+5, { }>})

Eval((g n, g m), {n → 4, m → 6, g→<n→n+5, { }>,
f→<n→n+5, { }>}) =
Eval((g n, g m), ρ') =>
Eval((g n, Eval(g m, ρ')), ρ') =>
Eval((g n, Eval(g (Eval (m, ρ')), ρ')), ρ') =>
Eval((g n, Eval(g (Val 6), ρ')), ρ') =>
Eval((g n, Eval((Eval(g, ρ'))(Val 6), ρ')), ρ') =>

Evaluate f (4,6);;

Let ρ' = {n → 4, m → 6, g→<n→n+5, { }>,
f→<n→n+5, { }>})

Eval((g n, Eval((Eval(g, ρ'))(Val 6), ρ')), ρ') =>
Eval((g n, Eval((Val<n→n+5,{ }>)(Val 6), ρ')), ρ') =>
Eval((g n, Eval(n+5, {n→6}+{ })), ρ') =
Eval((g n, Eval(n+5, {n→6})), ρ') =>
Eval((g n, Eval(n+(Eval(5, {n→6})), {n→6})), ρ') =>
Eval((g n, Eval(n+(Val 5), {n→6})), ρ') =>
Eval((g n, Eval((Eval(n,{n→6}))+(Val 5),{n→6}), ρ')=>
Eval((g n, Eval((Val 6)+(Val 5),{n→6}), ρ')=>

Evaluate f (4,6);;

Let ρ' = {n → 4, m → 6, g→<n→n+5, { }>,
f→<n→n+5, { }>})

Eval((g n, Eval((Val 6)+(Val 5),{n→6}), ρ') =>
Eval((g n, Val 11), ρ') =>
Eval((Eval(g n, ρ'), Val 11), ρ') =>
Eval((Eval(g (Eval(n, ρ')), ρ'), Val 11), ρ') =>
Eval((Eval(g (Val 4), ρ'), Val 11), ρ') =>
Eval((Eval(Eval(g, ρ')(Val 4), ρ'), Val 11), ρ') =>
Eval((Eval((Val<n→n+5, { }>)(Val 4), ρ'), Val 11), ρ')
=>

Evaluate f (4,6);;

Let ρ′ = {n → 4, m → 6, g→<n→n+5, { }>,
f→<n→n+5, { }>})

Eval((Eval((Val<n→n+5, { }>)(Val 4), ρ′), Val 11), ρ′) =>

Eval((Eval(n+5, {n → 4}+{})), Val 11), ρ′) =

Eval((Eval(n+5, {n → 4})), Val 11), ρ′) =>

Eval((Eval(n+Eval(5,{n → 4}),{n → 4}), Val 11),ρ′) =>

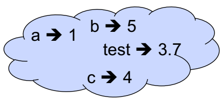
Eval((Eval(n+(Val 5),{n → 4}), Val 11),ρ′) =>

Eval((Eval(Eval(n,{n → 4})+(Val 5),{n → 4}),
Val 11),ρ′) =>

End of
Extra Material for Extra Credit

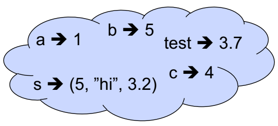
Tuples as Values

// ρ_7 = {c → 4, test → 3.7,
a → 1, b → 5}



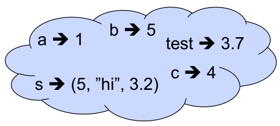
let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

// ρ_8 = {s → (5, "hi", 3.2),
c → 4, test → 3.7,
a → 1, b → 5}

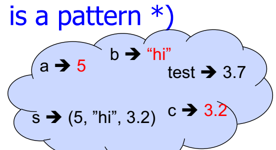


Pattern Matching with Tuples

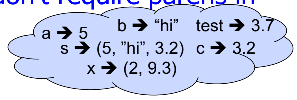
/ ρ_8 = {s → (5, "hi", 3.2),
c → 4, test → 3.7,
a → 1, b → 5}



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



let x = 2, 9.3;; (* tuples don't require parens in Ocaml *)
val x : int * float = (2, 9.3)



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;; (* _ matches all, binds nothing *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"

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")

Curried vs Uncurried

- Recall

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

```
# 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>
```


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

Recursive Functions

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


Recursion Example

Compute n^2 recursively using:
$n^2 = (2 * n - 1) + (n - 1)^2$

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

Structure of recursion similar to inductive proof

Recursion and Induction

```
# let rec nthsq n = match n with 0 -> 0
      | n -> (2 * n - 1) + nthsq (n - 1) ;;
```

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- **if** or **match** must contain base case
- Failure of these may cause failure of termination

Lists

- List can take one of two forms:
  - Empty list, written [ ]
  - Non-empty list, written x :: xs
    - x is head element, xs is tail list, :: called "cons"
  - Syntactic sugar: [x] == x :: [ ]
  - [x1; x2; …; xn] == x1 :: x2 :: … :: xn :: [ ]

Lists

```
# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1;
  1]
```


Lists are Homogeneous

```
# let bad_list = [1; 3.2; 7];;
Characters 19-22:
  let bad_list = [1; 3.2; 7];;
                      ^^^
This expression has type float but is here
  used with type int
```


Question

- Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [["hi"; "there"]; ["wahcha"]; [ ]; ["doin"]]

Answer

- Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [["hi"; "there"]; ["wahcha"]; [ ]; ["doin"]]

- 3 is invalid because of last pair

Functions Over Lists

```
# let rec double_up list =
    match list
    with [ ] -> [ ]  (* pattern before ->,
                        expression after *)
       | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>
# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1;
  1; 1; 1]
```

Functions Over Lists

```
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
# let rec poor_rev list =
  match list
  with [] -> []
    | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
# poor_rev silly;;
- : string list = ["there"; "there"; "hi"; "hi"]
```


Structural Recursion

- Functions on recursive datatypes (eg lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function

Question: Length of list

- Problem: write code for the length of the list
  - How to start?

```
let rec length list =
```


Question: Length of list

- Problem: write code for the length of the list
  - How to start?

```
let rec length list =
    match list with
```


Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```
let rec length list =
    match list with
```


Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```
let rec length list =
    match list with [] ->
     | (a :: bs) ->
```

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is empty?

```
let rec length list =
   match list with [] -> 0
    | (a :: bs) ->
```


Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is not empty?

```
let rec length list =
   match list with [] -> 0
    | (a :: bs) ->
```


Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is not empty?

```
let rec length list =
   match list with [] -> 0
    | (a :: bs) -> 1 + length bs
```


Structural Recursion : List Example

```
# let rec length list = match list
  with [ ] -> 0   (* Nil case *)
  | a :: bs -> 1 + length bs;;  (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [ ] is base case
- Cons case recurses on component list bs

Same Length

- How can we efficiently answer if two lists have the same length?

Same Length

- How can we efficiently answer if two lists have the same length?

```
let rec same_length list1 list2 =
   match list1 with [] ->
     (match list2 with [] -> true
       | (y::ys) -> false)
   | (x::xs) ->
     (match list2 with [] -> false
       | (y::ys) -> same_length xs ys)
```

Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```
let rec doubleList list =
```


Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```
let rec doubleList list =
  match list
   with [] ->[]
     | x :: xs -> (2 * x) :: doubleList xs
```


Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```
let rec doubleList list =
  match list
   with [] ->[]
     | x :: xs -> (2 * x) :: doubleList xs
```


Higher-Order Functions Over Lists

```
# let rec map f list =
  match list
  with [] -> []
  | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```


Higher-Order Functions Over Lists

```
# let rec map f list =
  match list
  with [] -> []
  | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```


Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# let doubleList list =
   List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# let doubleList list =
    List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

- Same function, but no explicit recursion

Folding Recursion

- Another common form “folds” an operation over the elements of the structure

```
# let rec multList list = match list
  with [ ] -> 1
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

- Computes (2 * (4 * (6 * 1)))

Folding Recursion : Length Example

```
# let rec length list = match list
  with [ ] -> 0   (* Nil case *)
  | a :: bs -> 1 + length bs;;  (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [ ] is base case, 0 is the base value
- Cons case recurses on component list bs
- What do multList and length have in common?