

# Programming Languages and Compilers (CS 421)

Elsa L Gunter  
2112 SC, UIUC



<https://courses.engr.illinois.edu/cs421/fa2024>

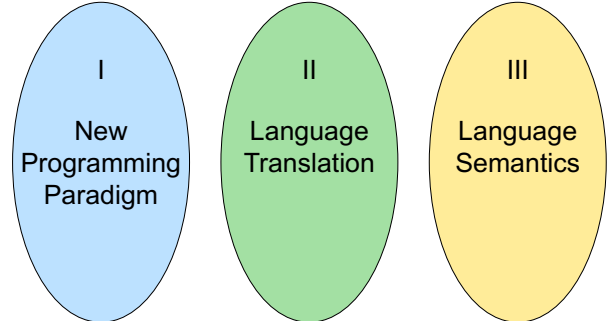
Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

8/27/24

1

## Programming Languages & Compilers

Three Main Topics of the Course

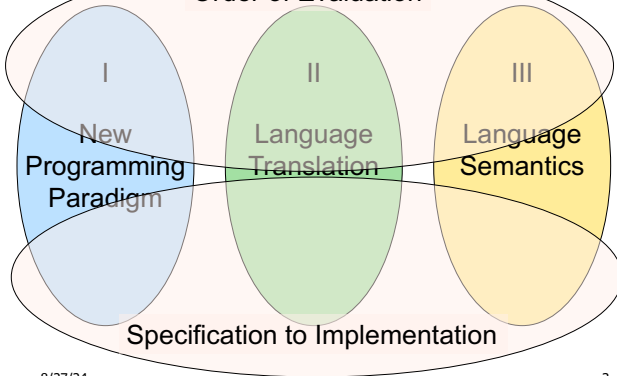


8/27/24

2

## Programming Languages & Compilers

Order of Evaluation

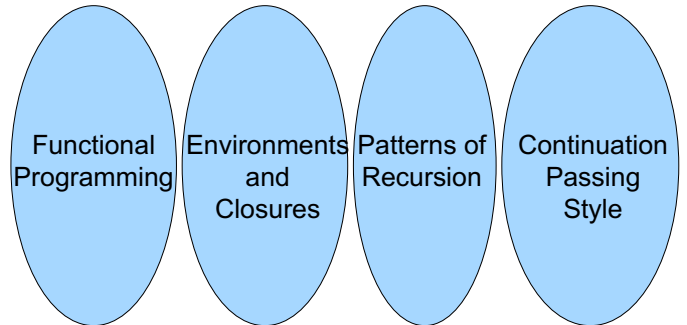


8/27/24

3

## Programming Languages & Compilers

I : New Programming Paradigm

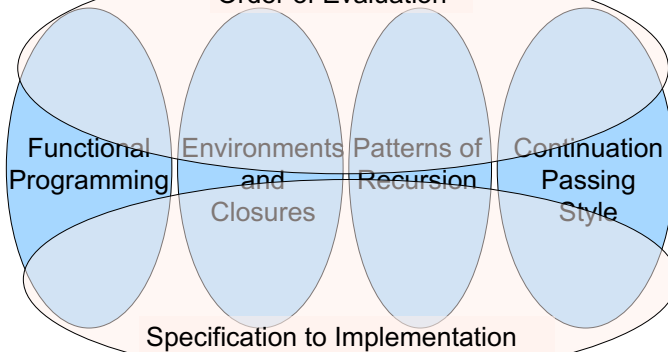


8/27/24

4

## Programming Languages & Compilers

Order of Evaluation

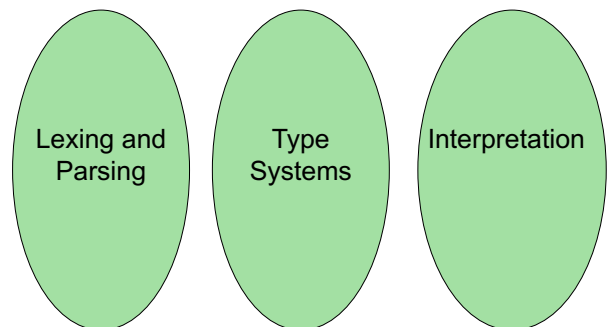


8/27/24

5

## Programming Languages & Compilers

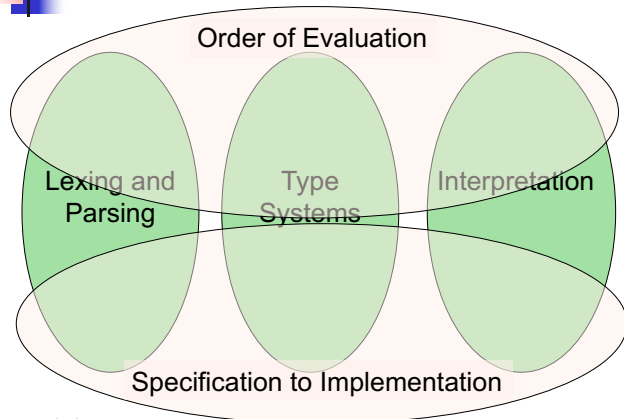
II : Language Translation



8/27/24

6

## Programming Languages & Compilers

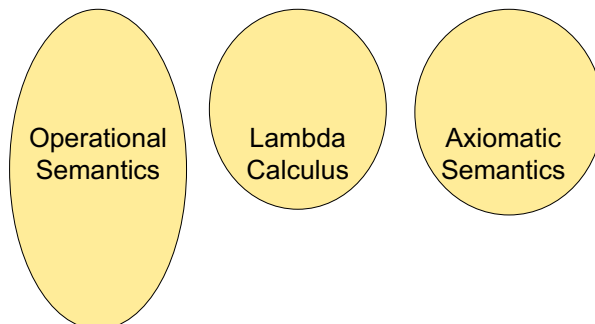


8/27/24

8

## Programming Languages & Compilers

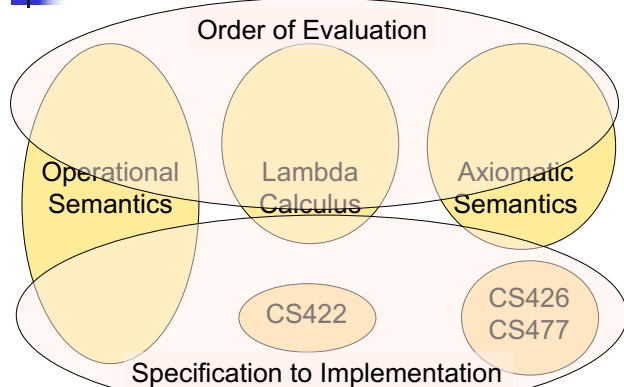
### III : Language Semantics



8/27/24

9

## Programming Languages & Compilers



8/27/24

10

## Contact Information - Elsa L Gunter

- Office: 2112 SC
- Office hours:
  - TBD
    - Can attend in zoom
  - Also by appointment
- Email: [egunter@illinois.edu](mailto:egunter@illinois.edu)

8/27/24

13

## Course TAs



Shams Alshabani



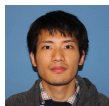
Athena Fung



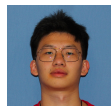
Helen Ilkiu



Paul Krogmeier



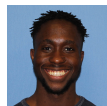
Yerong Li



Siheng Pan



Cody Rivera



Uche Uche-Ike



Allison Ye

8/27/24

14

## Course Website

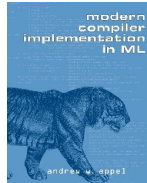
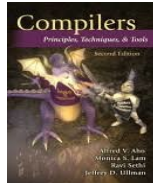
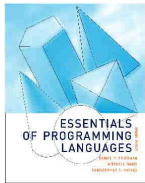
- <https://courses.engr.illinois.edu/cs421/sp2024>
- Main page - summary of news items
- Policy - rules governing course
- Lectures - syllabus and slides
- MPs - information about assignments
- Exams – Syllabi and review material for Midterms and finals
- Unit Projects - for 4 credit students
- Resources - tools and helpful info
- FAQ

8/27/24

15

## Some Course References

- No required textbook
- Some suggested references



8/27/24

16

## Some Course References

- No required textbook.
- Pictures of the books on previous slide
- Essentials of Programming Languages (2nd Edition) by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press 2001.
- Compilers: Principles, Techniques, and Tools, (also known as "The Dragon Book"); by Aho, Sethi, and Ullman. Published by Addison-Wesley. ISBN: 0-201-10088-6.
- Modern Compiler Implementation in ML by Andrew W. Appel, Cambridge University Press 1998
- Additional ones for Ocaml given separately

8/27/24

17

## Course Grading

- Assignments 14%
  - Web Assignments (WA) (~4-7%)
  - MPs (in Ocaml) (~5-8%)
  - All WAs and MPs Submitted by **PrairieLearn**
  - Late submission penalty:  
score capped at 80% of total

8/27/24

18

## Course Grading

- Three quizzes - 6% (2% each)
  - In class, BYOD
  - Tentatively Sep 17, Oct 8, Feb 27, Mar 26, Apr 23
- 3 Midterms - 15% each
  - **Sep 19-21, Oct 17-19, Nov 14-16**
  - **BE AVAILABLE FOR THESE DATES!**
- Final 35%
- CBTF Dec 12-15
- Percentages based on 3 cr, are approximate

8/27/24

19

## Course Assignments – WA & MP

- You may discuss assignments and their solutions with others
- You may work in groups, but you must **list members with whom you worked** if you share solutions or detailed solution outlines
- **Each student must write up and turn in their own solution separately**
- You may look at examples from class and other similar examples from any source – **cite appropriately**
  - Note: University policy on plagiarism still holds - cite your sources if you are not the sole author of your solution
  - Do not have to cite course notes or course staff

8/27/24

20

## OCAML

- Locally:
  - Will use ocaml inside VSCode inside PrairieLearn problems this semester
- Globally:
  - Main OCAML home: <http://ocaml.org>
  - To install OCAML on your computer see: <http://ocaml.org/docs/install.html>
  - To try on the web: <https://try.ocamlpro.com>
  - More notes on this later

8/27/24

21

## References for OCaml

- Supplemental texts (not required):
  - The Objective Caml system release 4.07, by Xavier Leroy, online manual
  - Introduction to the Objective Caml Programming Language, by Jason Hickey
  - Developing Applications With Objective Caml, by Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano, on O' Reilly
    - Available online from course resources

8/27/24

22

## Features of OCAML

- Higher order applicative language
- Call-by-value parameter passing
- Modern syntax
- Parametric polymorphism
  - Aka structural polymorphism
- Automatic garbage collection
- User-defined algebraic data types

8/27/24

25

## Why learn OCAML?

- Many features not clearly in languages you have already learned
- Assumed basis for much research in programming language research
- OCAML is particularly efficient for programming tasks involving languages (eg parsing, compilers, user interfaces)
- Industrially Relevant:
  - Jane Street trades billions of dollars per day using OCaml programs
  - Major language supported at Bloomberg
- Similar languages: Microsoft F#, SML, Haskell, Scala

8/27/24

26

## Course TAs



Shams Alshabani



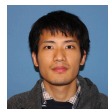
Athena Fung



Helen Ilkiu



Paul Krogmeier



Yerong Li



Siheng Pan



Cody Rivera



Uche Uche-Ike



Allison Ye

8/29/24

27

## Session in OCAML

```
% ocaml
Objective Caml version 4.07.1
# (* Read-eval-print loop; expressions and
  declarations *)
  2 + 3;; (* Expression *)
- : int = 5
# 3 < 2;;
- : bool = false
```

8/27/24

28

## Declarations; Sequencing of Declarations

```
# let x = 2 + 3;; (* declaration *)
val x : int = 5
# let test = 3 < 2;;
val test : bool = false
# let a = 1 let b = a + 4;; (* Sequence of dec
  *)
val a : int = 1
val b : int = 5
```

8/27/24

29

## Functions

```
# let plus_two n = n + 2;;  
val plus_two : int -> int = <fun>  
# plus_two 17;;  
- : int = 19
```

8/27/24

30

## Functions

```
let plus_two n = n + 2;;  
plus_two 17;;  
- : int = 19
```

8/27/24

32

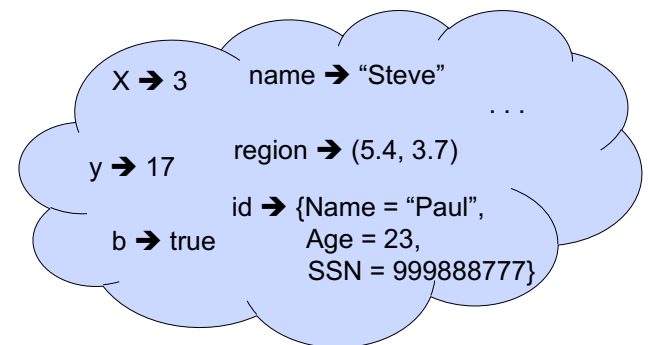
## Environments

- *Environments* record what value is associated with a given identifier
- Central to the semantics and implementation of a language
- Notation  
 $\rho = \{name_1 \rightarrow value_1, name_2 \rightarrow value_2, \dots\}$   
Using set notation, but describes a partial function
- Often stored as list, or stack
  - To find value start from left and take first match

8/27/24

45

## Environments



8/27/24

46

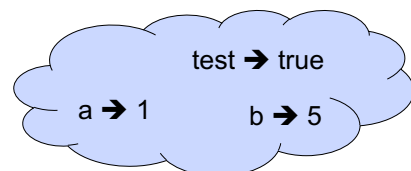
## Global Variable Creation

```
# 2 + 3;; (* Expression *)  
// doesn't affect the environment  
# let test = 3 < 2;; (* Declaration *)  
val test : bool = false  
//  $\rho_1 = \{test \rightarrow false\}$   
# let a = 1 let b = a + 4;; (* Seq of dec *)  
//  $\rho_2 = \{b \rightarrow 5, a \rightarrow 1, test \rightarrow false\}$ 
```

8/27/24

47

## Environments



8/27/24

48

## New Bindings Hide Old

```
// ρ2 = {b → 5, a → 1, test → false}  
let test = 3.7;;
```

- What is the environment after this declaration?

8/27/24

49

## New Bindings Hide Old

```
// ρ2 = {b → 5, a → 1, test → false}  
let test = 3.7;;
```

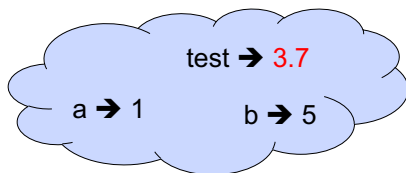
- What is the environment after this declaration?

```
// ρ3 = {test → 3.7, a → 1, b → 5}
```

8/27/24

50

## Environments



8/27/24

51

Now it's your turn

You should be able to do WA1-IC  
Problem 1 , parts (\* 1 \*) - (\* 3 \*)

8/27/24

52

## Local Variable Creation

```
// ρ3 = {test → 3.7, a → 1, b → 5}  
# let b = 5 * 4  
// ρ4 = {b → 20, test → 3.7, a → 1}  
in 2 * b;;  
- : int = 40  
// ρ5 = ρ3 = {test → 3.7, a → 1, b → 5}  
# b;;  
- : int = 5
```

Diagram illustrating environment frames for local variable creation. Three blue cloud-shaped frames are shown. The top frame (ρ<sub>3</sub>) contains 'a → 1', 'test → 3.7', and 'b → 5'. The middle frame (ρ<sub>4</sub>) contains 'a → 1', 'test → 3.7', and 'b → 20'. The bottom frame (ρ<sub>5</sub>) contains 'a → 1', 'test → 3.7', and 'b → 5'. Red arrows point from the middle and bottom frames up to the top frame, indicating they are children of ρ<sub>3</sub>.

8/27/24

54

## Local let binding

```
// ρ5 = ρ3 = {test → 3.7, a → 1, b → 5}  
# let c =  
  let b = a + a  
  // ρ6 = {b → 2} + ρ5  
  //   = {b → 2, test → 3.7, a → 1}  
  in b * b;;  
val c : int = 4  
// ρ7 = {c → 4, test → 3.7, a → 1, b → 5}  
# b;;  
- : int = 5
```

Diagram illustrating environment frames for local let binding. Three blue cloud-shaped frames are shown. The top frame (ρ<sub>5</sub>) contains 'a → 1', 'test → 3.7', and 'b → 5'. The middle frame (ρ<sub>6</sub>) contains 'a → 1', 'test → 3.7', and 'b → 2'. The bottom frame (ρ<sub>7</sub>) contains 'a → 1', 'test → 3.7', and 'b → 5'. Red arrows point from the middle and bottom frames up to the top frame, indicating they are children of ρ<sub>5</sub>.

8/27/24

55

## Local let binding

```
// ρ5 = ρ3 = {test → 3.7, a → 1, b → 5}
# let c =
  let b = a + a
  // ρ6 = {b → 2} + ρ3
  //   = {b → 2, test → 3.7, a → 1}
  in b * b;;
val c : int = 4
// ρ7 = {c → 4, test → 3.7, a → 1, b → 5}
# b;;
- : int = 5
```

8/27/24

56

## Local let binding

```
// ρ5 = ρ3 = {test → 3.7, a → 1, b → 5}
# let c =
  let b = a + a
  // ρ6 = {b → 2} + ρ3
  //   = {b → 2, test → 3.7, a → 1}
  in b * b;;
val c : int = 4
// ρ7 = {c → 4, test → 3.7, a → 1, b → 5}
# b;;
- : int = 5
```

8/27/24

57

## Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19
```

8/27/24

58

## Functions

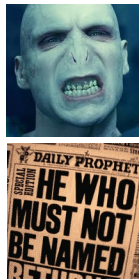
```
let plus_two n = n + 2;;
plus_two 17;;
- : int = 19
```

8/27/24

59

## Nameless Functions (aka Lambda Terms)

```
fun n -> n + 2;;
(fun n -> n + 2) 17;;
- : int = 19
```



8/27/24

60

## Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16
```

First definition syntactic sugar for second

8/27/24

62

## Using a nameless function

```
# (fun x -> x * 3) 5;; (* An application *)  
- : int = 15  
# ((fun y -> y +. 2.0), (fun z -> z * 3));;  
(* As data *)  
- : (float -> float) * (int -> int) = (<fun>, <fun>)
```

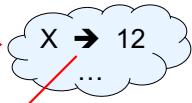
Note: in fun v -> exp(v), scope of variable is only the body exp(v)

8/27/24

63

## Values fixed at declaration time

```
# let x = 12;;  
val x : int = 12  
# let plus_x y = y + x;;  
val plus_x : int -> int = <fun>  
# plus_x 3;;
```



What is the result?

8/27/24

65

## Values fixed at declaration time

```
# let x = 12;;  
val x : int = 12  
# let plus_x y = y + x;;  
val plus_x : int -> int = <fun>  
# plus_x 3;;  
- : int = 15
```

8/27/24

66

## Values fixed at declaration time

```
# let x = 7;; (* New declaration, not an update *)  
val x : int = 7  
  
# plus_x 3;;
```

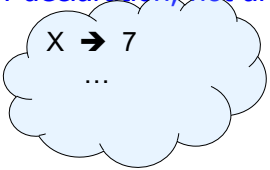
What is the result this time?

8/27/24

67

## Values fixed at declaration time

```
# let x = 7;; (* New declaration, not an update *)  
val x : int = 7  
# plus_x 3;;
```



What is the result this time?

8/27/24

68

## Values fixed at declaration time

```
# let x = 7;; (* New declaration, not an update *)  
val x : int = 7  
  
# plus_x 3;;  
- : int = 15
```

8/27/24

69



## Question

- Observation: Functions are first-class values in this language
- Question: What value does the environment record for a function variable?
- Answer: a closure

8/27/24

70

## Save the Environment!

- A *closure* is a pair of an environment and an association of a formal parameter (the input variables)\* with an expression (the function body), written:

$$f \rightarrow \langle (v_1, \dots, v_n) \rightarrow \text{exp}, \rho_f \rangle$$

- Where  $\rho_f$  is the environment in effect when  $f$  is defined (if  $f$  is a simple function)

\* Will come back to the "formal parameter"

8/27/24

71

## Closure for plus\_x

- When plus\_x was defined, had environment:

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

- Recall: `let plus_x y = y + x`

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

- Closure for `fun y -> y + x`:

$$\langle y \rightarrow y + x, \rho_{\text{plus\_x}} \rangle$$

- Environment just after plus\_x defined:

$$\{\text{plus\_x} \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus\_x}} \rangle\} + \rho_{\text{plus\_x}}$$

8/27/24

72

Now it's your turn

You should be able complete ACT1

8/27/24

73

## 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

8/27/24

78

## 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  $\rho_{\text{add\_three}}$  be the environment before the declaration  
■ Remember:  
let add_three =  
  fun x -> (fun y -> (fun z -> x + y + z));;  
Value:  $\langle x \rightarrow \text{fun } y \rightarrow (\text{fun } z \rightarrow x + y + z), \rho_{\text{add\_three}} \rangle$ 
```

8/27/24

79

## 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
```

8/27/24

80

## 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
```

8/27/24

81

## Functions as arguments

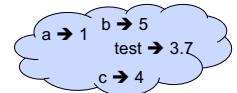
```
# let thrice f x = f (f (f x));;  
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>  
# let g = thrice plus_two;;  
val g : int -> int = <fun>  
# g 4;;  
- : int = 10  
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;  
- : string = "Hi! Hi! Hi! Good-bye!"
```

8/27/24

82

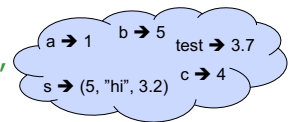
## Tuples as Values

```
//  $\rho_7 = \{c \rightarrow 4, \text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$ 
```



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

```
//  $\rho_8 = \{s \rightarrow (5, \text{"hi"}, 3.2), c \rightarrow 4, \text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$ 
```

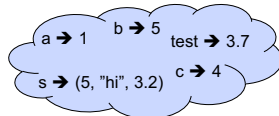


8/27/24

84

## Pattern Matching with Tuples

```
/  $\rho_8 = \{s \rightarrow (5, \text{"hi"}, 3.2), c \rightarrow 4, \text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$ 
```

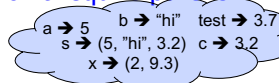
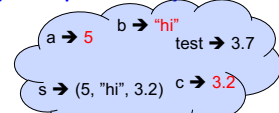


```
# let (a,b,c) = s;; (* (a,b,c) is a pattern *)
```

```
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)
```



8/27/24

85

## 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"
```

8/27/24

87

## 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")
```

8/27/24

88

## 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

8/27/24

89

## Closure for plus\_pair

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

- Closure for `plus_pair`:

$\langle (n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle$

- Environment just after `plus_pair` defined:

$\{\text{plus\_pair} \rightarrow \langle (n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} \rangle\}$   
+  $\rho_{\text{plus\_pair}}$

8/27/24

91

## Save the Environment!

- A *closure* is a pair of an environment and an association of a pattern (e.g.  $(v_1, \dots, v_n)$  giving the input variables) with an expression (the function body), written:

$\langle (v_1, \dots, v_n) \rightarrow \text{exp}, \rho \rangle$

- Where  $\rho$  is the environment in effect when the function is defined (for a simple function)

8/27/24

92

## Evaluating declarations

- Evaluation uses an environment  $\rho$
- To evaluate a (simple) declaration `let x = e`
  - Evaluate expression `e` in  $\rho$  to value `v`
  - Update  $\rho$  with `x v`:  $\{x \rightarrow v\} + \rho$
- Update:  $\rho_1 + \rho_2$  has all the bindings in  $\rho_1$  and all those in  $\rho_2$  that are not rebound in  $\rho_1$   
 $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{"hi"}\} + \{y \rightarrow 100, b \rightarrow 6\}$   
 $= \{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{"hi"}, b \rightarrow 6\}$

8/27/24

93

## Evaluating expressions in OCaml

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

8/27/24

94

## 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  $\{x \rightarrow v\} + \rho$
- 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  $e2$

8/27/24

95

## Evaluation of Application with Closures

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

8/27/24

96