Programming Languages and Compilers (CS 421)



Elsa L Gunter 2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

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

8/28/12



Contact Information - Elsa L Gunter

Office: 2112 SC

Office hours:

Mondays 11:00am – 11:50am

■ Tuesdays 3:30pm - 4:20pm

■ Thursdays 12:30pm - 1:40pm

Also by appointment

Email: egunter@illinois.edu

8/28/12 2



Contact Information - TAs

Teaching Assistants Office: 0207 SC

Kvle Blocher

■ Email: blocher1@illinois.edu

Hours: Tues 1:00pm – 1:50pm &
 Thurs 11:00am – 11:50am

Hassan Samee

■ Email: samee1@illinois.edu

Hours: Wed 1:30pm – 2:20pm &
 Fri 4:00pm - 4:50pm

8/28/12



Course Website

- http://courses.engr.illinois.edu/cs421
- Main page summary of news items
- Policy rules governing course
- Lectures syllabus and slides
- MPs information about homework
- Exams
- Unit Projects for 4 credit students
- Resources tools and helpful info
- FAQ

8/28/12 4



Some Course References

- No required textbook.
- 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

8/28/12



5

Course Grading

- Homework 20%
 - About 12 MPs (in Ocaml) and 12 written assignments
 - Submitted by handin on EWS linux machines
 MPs plain text code that compiles; HWs pdf
 - Late submission penalty: 20% of assignments total value
- 2 Midterms 20% each
 - In class Oct 9, Nov 13
 - DO NOT MISS EXAM DATES!
- Final 40% Dec 14, 7:00pm 10:00pm
- Percentages are approximate
 - Exams may weigh more if homework is much better



Course Homework

- You may discuss homeworks and their solutions with others
- You may work in groups, but you must list members with whom you worked if you share solutions or solution outlines
- Each student must turn in their own solution separately
- You may look at examples from class and other similar examples from any source
 - Note: University policy on plagiarism still holds cite your sources if you are not the sole author of your solution
- Problems from homework may appear verbatim, or with some modification on exams

8/28/12



Course Objectives

- New programming paradigm
 - Functional programming
 - Tail Recursion
 - Continuation Passing Style
- Phases of an interpreter / compiler
 - Lexing and parsing
 - Type checking
 - Evaluation
- Programming Language Semantics
 - Lambda Calculus
 - Operational Semantics

8/28/12 8



OCAML

- Compiler is on the EWS-linux systems at
- /usr/local/bin/ocaml
- A (possibly better, non-PowerPoint) text version of this lecture can be found at
- http://course.engr.illinois.edu/class/cs421/ lectures/ocaml-intro-shell.txt
- For the OCAML code for today's lecture see
- http://course.engr.illinois.edu/class/cs421/ lectures/ocaml-intro.ml

8/28/12



WWW Addresses for OCAML

- Main CAML home: http://caml.inria.fr/index.en.html
- To install OCAML on your computer see:
- http://caml.inria.fr/ocaml/release.en.html

8/28/12 10



References for CAML

- Supplemental texts (not required):
- The Objective Caml system release 3.09, 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

8/28/12 11



OCAML

- CAML is European descendant of original ML
 - American/British version is SML
 - O is for object-oriented extension
- ML stands for Meta-Language
- ML family designed for implementing theorem provers
 - It was the meta-language for programming the "object" language of the theorem prover
 - Despite obscure original application area, OCAML is a full general-purpose programming language



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
- It's fast winners of the 1999 and 2000 ICFP Programming Contests used OCAML

8/28/12 13



- 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)
- Used at Microsoft for writing SLAM, a formal methods tool for C programs

8/28/12 14



Session in OCAML

% ocaml

Objective Caml version 3.12.0

(* Read-eval-print loop; expressions and declarations *)

(* Expression *) 2 + 3::

-: int = 5

3 < 2;;

-: bool = false

8/28/12



No Overloading for Basic Arithmetic Operations

```
# 15 * 2;;
-: int = 30
# 1.35 + 0.23;; (* Wrong type of addition *)
Characters 0-4:
 1.35 + 0.23;; (* Wrong type of addition *)
Error: This expression has type float but an expression was expected of type
# 1.35 +. 0.23;;
-: float = 1.58
```

8/28/12 16



No Implicit Coercion

```
# 1.0 * 2;; (* No Implicit Coercion *)
Characters 0-3:
 1.0 * 2;; (* No Implicit Coercion *)
```

Error: This expression has type float but an expression was expected of type

8/28/12

17

15



Sequencing Expressions

```
# "Hi there";; (* has type string *)
-: string = "Hi there"
# print string "Hello world\n";; (* has type unit *)
Hello world
-: unit =()
# (print_string "Bye\n"; 25);; (* Sequence of exp *)
-: int = 25
```

8/28/12 18



Terminology

- Output refers both to the result returned from a function application
 - As in + outputs integers, whereas +. outputs floats
- And to text printed as a side-effect of a computation
 - As in print string "\n" outputs a carriage return
 - In terms of values, it outputs () ("unit")
- We will standardly use "output" to refer to the value returned

8/28/12 19



Declarations; Sequencing of Declarations

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

8/28/12 20



Environments

- Environments record what value is associated with a given identifier
- Central to the semantics and implementation of a language
- Notation

 $\rho = \{ \mathsf{name}_1 \rightarrow \mathsf{value}_1, \, \mathsf{name}_2 \rightarrow \mathsf{value}_2, \, ... \}$ Using set notation, but describes a partial function

- Often stored as list, or stack
 - To find value start from left and take first match

8/28/12 21



Global Variable Creation

```
# 2 + 3;; (* Expression *)

// doesn't effect 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}
```

8/28/12

22



New Bindings Hide Old

//
$$\rho_2 = \{b \rightarrow 5, a \rightarrow 1, \text{ test } \rightarrow \text{ false}\}\$$
 let $a = 3;;$

What is the environment after this declaration?

4

23

New Bindings Hide Old

//
$$\rho_2 = \{b \rightarrow 5, a \rightarrow 1, \text{ test } \rightarrow \text{ false}\}\$$
 let $a = 3;;$

What is the environment after this declaration?

//
$$\rho_3 = \{a \rightarrow 3, b \rightarrow 5, \text{ test } \rightarrow \text{ false}\}$$

8/28/12 24

8/28/12

```
Local let binding

// \rho_3 = \{a \rightarrow 3, b \rightarrow 5, \text{ test } \rightarrow \text{ false}\}

# let c =

let b = a + a

// \rho_4 = \{b \rightarrow 6\} + \rho_2

// = \{b \rightarrow 6, a \rightarrow 3, \text{ test } \rightarrow \text{ false}\}

in b * b;;

val c : \text{ int } = 36

// \rho_5 = \{c \rightarrow 36, a \rightarrow 3, b \rightarrow 5, \text{ test } \rightarrow \text{ false}\}
```

25

b;;

-: int = 5

8/28/12

```
Local Variable Creation

// \rho_5 = \{c \rightarrow 36, b \rightarrow 5, a \rightarrow 3, \text{ test } \rightarrow \text{ false}\}

# let b = 5 * 4

// \rho_6 = \{b \rightarrow 20, c \rightarrow 36, a \rightarrow 3, \text{ test } \rightarrow \text{ false}\}

in 2 * b;;

-: int = 40

// \rho_7 = \rho_5

# b;;

-: int = 5
```

```
# true;;
-: bool = true
# false;;
-: bool = false

# if y > x then 25 else 0;;
-: int = 25
```

```
# 3 > 1 && 4 > 6;;
-: bool = false
# 3 > 1 || 4 > 6;;
-: bool = true
# (print_string "Hi\n"; 3 > 1) || 4 > 6;;
Hi
-: bool = true
# 3 > 1 || (print_string "Bye\n"; 4 > 6);;
-: bool = true
# not (4 > 6);;
-: bool = true
```

```
# let s = (5,"hi",3.2);;

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

# 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)
```

```
# (*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"
```

```
# 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
```

```
# (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)
```

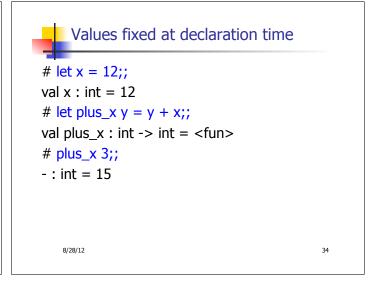
Values fixed at declaration time

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?

8/28/12

33

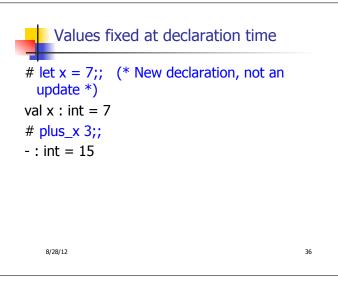


Values fixed at declaration time
let x = 7;; (* New declaration, not an
 update *)
val x : int = 7
plus_x 3;;

What is the result this time?

8/28/12

35



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
```

8/28/12



Partial application of functions

let add_three x y z = x + y + z;;

8/28/12

38



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!"
```

8/28/12 39



37

41

Question

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

8/28/12 40



Save the Environment!

 A closure is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:

$$f \rightarrow \langle (v1,...,vn) \rightarrow exp, \rho_f \rangle$$

Where ρ_f is the environment in effect when f is defined (if f is a simple function)

8/28/12



Closure for plus_x

When plus x was defined, had environment:

$$\rho_{\text{plus } X} = \{x \to 12, ..., y \to 24, ...\}$$

Closure for plus_x:

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

Environment just after plus_x defined:

$$\{plus_x \rightarrow \langle y \rightarrow y + x, \rho_{plus_x} \rangle\} + \rho_{plus_x}$$



Evaluation of Application of plus_x;;

Have environment:

$$\rho = \{ plus_x \rightarrow \langle y \rightarrow y + x, \rho_{plus_x} \rangle, \dots, \\ y \rightarrow 3, \dots \}$$

where $\rho_{\text{plus}_X} = \{x \rightarrow 12, ..., y \rightarrow 24, ...\}$

- Eval (plus_x y, ρ) rewrites to
- Eval (app $\langle y \rightarrow y + x, \rho_{plus_x} \rangle 3, \rho$) rewrites to
- Eval (y + x, {y \rightarrow 3} + ρ_{plus_x}) rewrites to
- Eval $(3 + 12, \rho_{\text{plus x}}) = 15$

8/28/12

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")
```

Match Expressions

let triple_to_pair triple =

•Each clause: pattern on left, expression on right

43

45

•Each x, y has scope of only its clause

•Use first matching clause

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

8/28/12



8/28/12

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:

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

Environment just after plus_pair defined:

$$\begin{aligned} \{ plus_pair \rightarrow < & (n,m) \rightarrow n + m, \; \rho_{plus_pair} > \} \\ & \quad + \; \rho_{plus_pair} \end{aligned}$$

8/28/12



Evaluation of Application with Closures

- In environment ρ , evaluate left term to closure, $c = \langle (x_1,...,x_n) \rightarrow b, \rho \rangle$
- (x₁,...,x_n) variables in (first) argument
- Evaluate the right term to values, (v₁,...,v_n)
- Update the environment o to

$$\rho' = \{x_1 \rightarrow v_1, ..., x_n \rightarrow v_n\} + \rho$$

Evaluate body b in environment ρ'

8/28/12 47



Evaluation of Application of plus_pair

Assume environment

$$\rho = \{x \to 3..., \\ plus_pair \to <(n,m) \to n + m, \rho_{plus_pair}>\} + \\ \rho_{plus_pair}$$

- $ρ_{\text{plus_pair}}$ Eval (plus_pair (4,x), ρ)=
- Eval (app <(n,m) \rightarrow n + m, $\rho_{\text{plus pair}}>$ (4,x), ρ)) =
- Eval (app <(n,m) \rightarrow n + m, $\rho_{plus\ pair}$ > (4,3), ρ)) =
- Eval $(n + m, \{n -> 4, m -> 3\} + \rho_{\text{plus pair}}) =$
- Eval $(4 + 3, \{n -> 4, m -> 3\} + \rho_{plus_pair}) = 7$

8/28/12

48

Curried vs Uncurried

Recall

val add_three : int -> int -> int -> int = <fun>

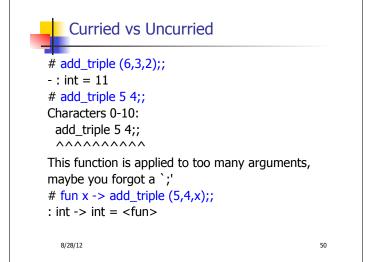
How does it differ from

```
# 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

8/28/12





Scoping Question

Consider this code:

```
let x = 27;;
let f x =
let x = 5 in
(fun x -> print_int x) 10;;
f 12;;
```

What value is printed?

5

10

12 27