

Programming Languages and Compilers (CS 421)



William Mansky

<http://courses.engr.illinois.edu/cs421/su2013/>

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



Contact Information – William Mansky

- Office: 0207 SC
- Office hours:
 - After every class: 12:45PM-1:45PM
 - Online Tuesday 7PM-8PM
 - Also by appointment
- Email: mansky1@illinois.edu
- Skype: william.mansky



Contact Information - TAs

- Teaching Assistants' Office: 0207 SC
- Prasanna Giridhar
 - Email: giridha2@illinois.edu
 - Skype: TBA
 - Hours: TBA



Course Website

- <http://courses.engr.illinois.edu/cs421/su2013/>
- 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

- Class discussion board on <http://piazza.com>
- Post questions about course, assignments, exams, etc.
- Instructor and TA check once/day
- Collaboration on HWs/MPs is allowed, but you must ***write and test your code separately*** (see policy for details)



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



Course Grading

- Homework 20%
 - About 9 MPs (in OCaml) and 8 written assignments
 - MPs submitted by **handin** on EWS linux machines
 - Late submission penalty: 20% of assignments total value
- 2 Midterms - 20% each
 - In class – **June 27, July 18**
- **DO NOT MISS EXAM DATES!**
- Final 40% - Aug 4 1PM – 3PM
- 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



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

- Compiler is on the EWS-linux systems
- A (possibly better, non-PowerPoint) text version of this lecture can be found at
- <http://courses.engr.illinois.edu/cs421/su2013/lectures/ocaml-intro-shell.txt>
- For the OCaml code for today's lecture see
- <http://www.cs.illinois.edu/class/cs421/lectures/ocaml-intro.ml>



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>



References for CAML

- Supplemental texts (not required):
- The Objective Caml system release 4.00, by Xavier Leroy et al., 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

- 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
- Despite obscure original application area, OCaml is a full general-purpose programming language



Features of OCaml

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



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 (e.g. parsing, compilers, user interfaces)
- Used at Microsoft for writing SLAM, a formal methods tool for C programs



Session in OCaml

```
% ocaml
```

```
Objective Caml version 4.00
```

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

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

```
- : int = 5
```

```
# 3 < 2;;
```

```
- : bool = false
```




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
int

```
# 1.35 +. 0.23;;
```

```
- : float = 1.58
```



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
int



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 *)
```

```
Bye
```

```
- : int = 25
```



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



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
```



Environments

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

$$\rho = \{\text{name}_1 \rightarrow \text{value}_1, \text{name}_2 \rightarrow \text{value}_2, \dots\}$$

Using set notation, but describes a partial function

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



Global Variable Creation

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

```
// doesn't effect the environment
```

```
# let test = 3 < 2;;    (* Declaration *)
```

```
val test : bool = false
```

```
//  $\rho = \{\text{test} \rightarrow \text{false}\}$ 
```

```
# let a = 3 let b = a + 2;; (* Sequence of dec  
*)
```

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



Local let binding

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

```
# let b = 5 * 4 in 2 * b;;
```

```
- : int = 40
```

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

```
# let c =
```

```
    let b = a + a
```

```
    in b * b;;
```

```
val c : int = 36
```

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

```
# b;;
```

```
- : int = 5
```




Local Variable Creation

```
# let c =
```

```
  let b = a + a
```

```
//  $\rho_1 = \{b \rightarrow 6, a \rightarrow 3, \text{test} \rightarrow \text{false}\}$ 
```

```
  in b * b;;
```

```
val c : int = 36
```

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

```
# b;;
```

```
- : int = 5
```



Booleans (aka Truth Values)

```
# true;;
```

```
- : bool = true
```

```
# false;;
```

```
- : bool = false
```

```
# if y > x then 25 else 0;;
```

```
- : int = 25
```



Booleans

```
# 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
```



Tuples

```
# 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

(*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

```
# 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



Using an anonymous 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)`



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?



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
```



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?



Values fixed at declaration time

let x = 7;; (* New declaration, not an
update *)

val x : int = 7

plus_x 3;;

- : int = 15



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 =
```

```
  match triple
```

```
  with (0, x, y) -> (x, y)
```

```
  | (x, 0, y) -> (x, y)
```

```
  | (x, y, _) -> (x, y);;
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

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



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
```



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*



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
```




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>
```



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!"
```



Question

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



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:

$$f \rightarrow < (v_1, \dots, v_n) \rightarrow \text{exp}, \rho_f >$$

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



Closure for plus_x

- When plus_x was defined, had environment:

$$\rho_{\text{plus_x}} = \{x \rightarrow 12, \dots, y \rightarrow 24, \dots\}$$

- Closure for plus_x:

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

- Environment just after plus_x defined:

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



Combining Environments

- We combine environments with +
- Almost like set union
- Conflicts are resolved in a left-biased manner
 - $\{y \rightarrow 3, x \rightarrow 7\} + \{y \rightarrow 9, \dots\} = \{y \rightarrow 3, x \rightarrow 7, \dots\}$



Evaluation of Application with Closures

- In environment ρ , 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
- Evaluate the right term to values, (v_1, \dots, v_n)
- Update the environment ρ to
 $\rho' = \{x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n\} + \rho$
- Evaluate body b in environment ρ'

Evaluation of Application of plus_x;;

- Have environment:

$$\rho = \{\text{plus_x} \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle, \dots, \\ y \rightarrow 3, x \rightarrow 7, \dots\}$$

where $\rho_{\text{plus_x}} = \{x \rightarrow 12, \dots, y \rightarrow 24, \dots\}$

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



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



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