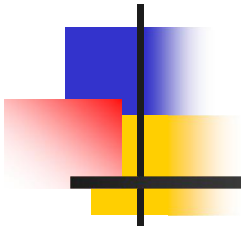


Programming Languages and Compilers (CS 421)



Reza Zamani

<http://www.cs.illinois.edu/class/cs421/>

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



Contact Information – Reza Zamani

- n Email: zamani@illinois.edu
- n Office hours will be announced on the course webpage.



Contact Information - Choonghwan Lee

- n Email: cleee83@illinois.edu
- n Office hours will be announced on the course webpage.



Course Website

- n Main page - summary of news items
- n Policy - rules governing course
- n Lectures - syllabus and slides
- n MPs - information about homework
- n Exams
- n Unit Projects - for 4 credit students
- n Resources - tools and helpful info
- n FAQ



Newsgroup

- n You **have to** read the newsgroup frequently; it is part of the instruction. Some important clarifications, notifications, ... will appear only there.
- n Setup your newsreader to read the news feed "**class.su11.cs421**" at "news.cs.illinois.edu".
- n For help to setup your newsreader, see:
<http://agora.cs.illinois.edu/display/tsg/Newsreader+Documentation>



Some Course References

- n No required textbook.
- n Essentials of Programming Languages (2nd Edition) by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press 2001.
- n 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.
- n Modern Compiler Implementation in ML by Andrew W. Appel, Cambridge University Press 1998
- n Additional ones for Ocaml given separately



Course Grading

- n Homework 25%
 - n About 7 MPs (in Ocaml) and maybe 1 written assignment
 - n MPs submitted by **handin** on EWS linux machines
 - n HWs turned in in class
 - n Late submission penalty: 20% of assignments total value
- n Midterms - 30%
 - n In class –**Jul 14**
- n **DO NOT MISS EXAM DATES!**
- n Final 45% - time/date will be annouced on the course webpage.



Course Homework

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



Course Objectives

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



- n Compiler is on the EWS-linux systems at
/usr/local/bin/ocaml



WWW Addresses for OCAML

n Main CAML home:

<http://caml.inria.fr/index.en.html>

n To install OCAML on your computer see:

<http://caml.inria.fr/ocaml/release.en.html>

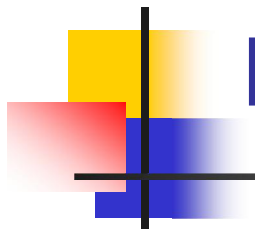


References for CAML

- n Supplemental texts (not required):
 - n The Objective Caml system release 3.09, by Xavier Leroy, online manual
 - n Introduction to the Objective Caml Programming Language, by Jason Hickey
 - n Developing Applications With Objective Caml, by Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano, on O'Reilly
 - n Available online from course resources



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



Features of OCAML

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

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





Why learn OCAML?

- n Many features not clearly in languages you have already learned
- n Assumed basis for much research in programming language research
- n OCAML is particularly efficient for programming tasks involving languages (eg parsing, compilers, user interfaces)
- n Used at Microsoft for writing SLAM, a formal methods tool for C programs



Session in OCAML

```
% ocaml
```

```
Objective Caml version 3.12.0
```

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

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

```
- : int = 5
```

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

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



Session in OCAML

- n You may want to use with *ledit*
% `ledit ocaml`
to get better editing capabilities.



Environments

- n *Environments* record what value is associated with a given variable
- n Central to the semantics and implementation of a language
- n 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
- n Often stored as list, or stack
- n To find value start from left and take first match



Sequencing

```
# "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
```

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

```
val a : int = 3
```

```
val b : int = 5
```



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

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

```
- : int = 40
```

```
//      = {b      5, a      3, test      false}
```

```
# let c =
```

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

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

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

```
//      = {c      36, b      5, a      3, test      false}
```

```
# b;;
```

```
- : int = 5
```



Local Variable Creation

```
# let c =
```

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

```
//    1 = {b    5, a    3, test    false}
```

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

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

```
//    = {c    36, b    5, a    3, test    false}
```

```
# b;;
```

```
- : int = 5
```



Terminology

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



No Overloading for Basic Arithmetic Operations

```
# let x = 5 + 7;;
```

```
val x : int = 12
```

```
# let y = x * 2;;
```

```
val y : int = 24
```

```
# let z = 1.35 + 0.23;; (* Wrong type of addition *)
```

Characters 8-12:

```
let z = 1.35 + 0.23;; (* Wrong type of addition *)
```

^ ^ ^ ^

This expression has type float but is here used with type int

```
# let z = 1.35 +. 0.23;;
```

```
val z : float = 1.58
```



No Implicit Coercion

```
# let u = 1.0 + 2;;
```

Characters 8-11:

```
let u = 1.0 + 2;;  
      ^ ^ ^
```

This expression has type float but is here used with
type int

```
# let w = y + z;;
```

Characters 12-13:

```
let w = y + z;;  
          ^
```

This expression has type float but is here used with
type int



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
```



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 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 `v` 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 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
```



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
```



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

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



Save the Environment!

- n 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:

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

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



Closure for plus_x

- n When plus_x was defined, had environment:

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

- n Closure for plus_x:

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

- n Environment just after plus_x defined:

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



Evaluation of Application

- n First evaluate the left term to a function (ie starts with fun)
- n Evaluate the right term (argument) to a value
 - n Things starting with fun are values
- n Substitute the argument for the formal parameter in the body of the function
- n Evaluate resulting term
- n (Need to use environments)



Evaluation Application of plus_x;;

n Have environment:

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

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

n Eval (plus_x y, ρ) rewrites to

n Eval (app $\langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle$ 3, ρ)
rewrites to

n Eval (3 + x, $\rho_{\text{plus_x}}$) rewrites to

n Eval (3 + 12, $\rho_{\text{plus_x}}$) = 15



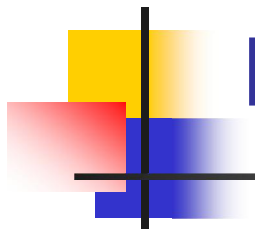
Scoping Question

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



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 *)  
   (* More on this later *)
```