Contact Information - Elsa L Gunter

- Office: 2112 SC
- Office hours:
  - Tuesday 12:00pm – 1:30pm
  - Thursday 3:30pm - 4:20pm
  - Also by appointment
- Email: egunter@illinois.edu
Course TAs

John Lee
Kevin Banker
Eric Huber
Tom Bogue
Deniz Arsan
Contact Information - TAs

- Teaching Assistants Office: 0207 SC
- John Lee
  - Email: lee170@illinois.edu
  - Hours: Mon 10:00am – 10:50am
    Fri 10:00am – 10:50am
- Tom Bogue
  - Email: tbogue2@illinois.edu
  - Hours: Wed 10:00am – 10:50am
    Fri 12:30pm – 1:20pm
Contact Information – TAs cont

- Kevin Banker
  - Email: banker2@illinois.edu
  - Hours: Tues 10:00am – 10:50am
  - Thurs 12:30pm – 1:20pm

- Eric Huber
  - Email: echuber2@illinois.edu
  - Hours: Mon 3:30pm – 4:20pm
  - Wed 1:00pm – 1:50pm
Contact Information – TAs cont

- Deniz Arsyan
  - Email: darsan2@illinois.edu
  - Hours: Mon 2:00pm – 2:50pm
    Wed 3:30pm – 4:20pm
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 assignments
- Exams
- Unit Projects - for 4 credit students
- Resources - tools and helpful info
- FAQ
Some Course References

- No required textbook
- Some suggested references

[Images of books: Essentials of Programming Languages, Compilers: Principles, Techniques, & Tools, Modern Compiler Implementation in ML]
Some Course References

- No required textbook.
- Pictures of the books on previous slide
- Additional ones for Ocaml given separately
Course Grading

- Assignments 20%
  - About 12 Web-based Assignments (WA) (~7%)
    - May be converted to Hand Written (HW) if our web development is inadequate
  - About 8 MPs (in Ocaml) (~7%)
  - About 4 Labs (~6%)
  - All MPs Submitted by `svn`
    - MPs – plain text code that compiles; HWs – pdf
  - Late submission penalty for MLs and WA: 20% of assignments total value

- Two Midterms (CBTF) %20 each
- Final (CBTF) %40
Computer-Based Testing Facility

- Labs and exams in Computer-Based Testing Center (Basement of Granger)
  - Self-scheduled
  - Over a four day period
  - No extensions beyond the four day period
  - Fall back:
    - Labs become MPs
    - Exams are given in class on the date marked for the exam
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 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
Three Main Topics of the Course

I. New Programming Paradigm
II. Language Translation
III. Language Semantics
Programming Languages & Compilers

Order of Evaluation

I
New Programming Paradigm

II
Language Translation

III
Language Semantics

Specification to Implementation
I : New Programming Paradigm

Functional Programming
Environments and Closures
Patterns of Recursion
Continuation Passing Style
Programming Languages & Compilers

Order of Evaluation

Functional Programming
Environments and Closures
Patterns of Recursion
Continuation Passing Style

Specification to Implementation
II : Language Translation

- Lexing and Parsing
- Type Systems
- Interpretation
III : Language Semantics

Operational Semantics

Lambda Calculus

Axiomatic Semantics
Course Objectives

- New programming paradigm
  - Functional programming
  - Environments and Closures
  - Patterns of Recursion
  - Continuation Passing Style

- Phases of an interpreter / compiler
  - Lexing and parsing
  - Type systems
  - Interpretation

- Programming Language Semantics
  - Lambda Calculus
  - Operational Semantics
  - Axiomatic Semantics
OCAML

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

Globally:
- Main CAML home: http://ocaml.org
- To install OCAML on your computer see: http://ocaml.org/docs/install.html
References for OCaml

- Supplemental texts (not required):
  - The Objective Caml system release 4.03, 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
OCAML Background

- 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, OCAMAL 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
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
- Similar languages: Microsoft F#, SML, Haskell, Scala
OCaml Intro Code

- 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](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](http://course.engr.illinois.edu/class/cs421/lectures/ocaml-intro.ml)
Session in OCAML

% ocaml

Objective Caml version 4.01

# (* 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
Declarations; Sequencing of Declarations

```ocaml
# 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
```
Environments

- Environments record what value is associated with a given identifier.
- Central to the semantics and implementation of a language.
- Notation:
  \[ \rho = \{\text{name}_1 \rightarrow \text{value}_1, \text{name}_2 \rightarrow \text{value}_2, \ldots\} \]
  Using set notation, but describes a partial function.
- Often stored as list, or stack.
  - To find value start from left and take first match.
Environments

X \Rightarrow 3

name \Rightarrow “Steve”

y \Rightarrow 17

region \Rightarrow (5.4, 3.7)

b \Rightarrow \text{true}

id \Rightarrow \{Name = “Paul”,
            Age = 23,
            SSN = 9998888777\}
Global Variable Creation

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

test $\rightarrow$ true

a $\rightarrow$ 1

b $\rightarrow$ 5
New Bindings Hide Old

// \( \rho_2 = \{b \rightarrow 5, a \rightarrow 1, \text{test} \rightarrow \text{false}\} \)

let test = 3.7;;

- What is the environment after this declaration?
New Bindings Hide Old

// $\rho_2 = \{b \rightarrow 5, a \rightarrow 1, \text{test} \rightarrow \text{false}\}$
let test = 3.7;;

- What is the environment after this declaration?

// $\rho_3 = \{\text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$
Now it’s your turn

You should be able to do WA1
Problem 1, parts (* 1 *) and (* 2 *)
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
Local let binding

// ρ₅ = {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
Local let binding

// \(\rho_5 = \{\text{test} \rightarrow 3.7, \ a \rightarrow 1, \ b \rightarrow 5\}\)
# let c =
    let b = a + a
// \(\rho_6 = \{b \rightarrow 2\} + \rho_3\)
//   =\{b \rightarrow 2, \ \text{test} \rightarrow 3.7, \ a \rightarrow 1\}\n    in b * b;;
val c : int = 4
// \(\rho_7 = \{c \rightarrow 4, \ \text{test} \rightarrow 3.7, \ a \rightarrow 1, \ b \rightarrow 5\}\)
# b;;
- : int = 5
Local let binding

// ρ₅ = {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
Now it’s your turn

You should be able to do WA1 Problem 1, parts (* 3 *) and (* 4 *)
Booleans (aka Truth Values)

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

// \( \rho_7 = \{c \rightarrow 4, \text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\} \)
# if b > a then 25 else 0;;
- : int = 25
Booleans and Short-Circuit Evaluation

# 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
Now it’s your turn

You should be able to do WA1 Problem 1, part (* 5 *)
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)

// \( \rho_8 = \{s \rightarrow (5, "hi", 3.2), \ c \rightarrow 4, \ \text{test} \rightarrow 3.7, \ a \rightarrow 1, \ b \rightarrow 5\} \)
Pattern Matching with Tuples

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

# 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)
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"
Now it’s your turn

You should be able to do WA1 through step
Functions

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

let plus_two n = n + 2;;

plus_two 17;;

- : int = 19
Nameless Functions (aka Lambda Terms)

```ml
fun n -> n + 2;;

(fun n -> n + 2) 17;;
- : int = 19
```
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 is only the body exp(v)
Values fixed at declaration time

```ocaml
# 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

```ocaml
# 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;;

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
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:

  \[ f \rightarrow < (v_1, \ldots, v_n) \rightarrow \text{exp}, \ \rho_f > \]

- Where \( \rho_f \) 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:
  \[ \rho_{\text{plus}_x} = \{ ..., x \rightarrow 12, ... \} \]
- Recall: let plus_x y = y + x
  is really let plus_x = fun y -> y + x
- Closure for fun y -> y + x:
  \[ <y \rightarrow y + x, \rho_{\text{plus}_x} > \]
- Environment just after plus_x defined:
  \[ \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x} > \} + \rho_{\text{plus}_x} \]
Now it’s your turn

You should be able to do HW1
Problem 1, parts (* 7 *) and (* 8 *)
Evaluation of Application of plus_x;;

- Have environment:
  \[ \rho = \{ \text{plus}_x \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle, \ldots, y \rightarrow 3, \ldots \} \]

  where \( \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)

- \( \text{Eval} \ (\text{plus}_x \ y, \ \rho) \) rewrites to

- \( \text{Eval} \ (\text{App} \ \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle > \ 3, \ \rho) \) rewrites to

- \( \text{Eval} \ (y + x, \ \{ y \rightarrow 3 \} + \rho_{\text{plus}_x}) \) rewrites to

- \( \text{Eval} \ (3 + 12, \ \rho_{\text{plus}_x}) = 15 \)
Functions with more than one argument

```ocaml
# 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
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

```ocaml
# 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!"
```
Functions on tuples

```ocaml
# 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")
```
# 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
Assume $\rho_{\text{plus\_pair}}$ was the environment just before \texttt{plus\_pair} defined.

Closure for \texttt{plus\_pair}:

\[
<(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}}>
\]

Environment just after \texttt{plus\_pair} defined:

\[
\{\text{plus\_pair} \rightarrow <(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} > \} + \rho_{\text{plus\_pair}}
\]