Contact Information - Elsa L Gunter
- Office: 2112 SC
- Office hours:
  - Tuesdays 12:30pm – 1:45pm
  - Thursdays 3:30pm – 4:45pm
  - Fridays 10:00am – 11:00am
- Also by appointment
- Email: egunter@illinois.edu

Contact Information - TAs
- Teaching Assistants Office: 0207 SC
- Susannah Johnson
  - Email: sjohnsn2@illinois.edu
  - Hours: Mon 2:00pm – 3:45pm
- Edgar Pek
  - Email: pek1@illinois.edu
  - Hours: Wed 10:00am – 11:45am
- Yet To Be Determined

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

Some Course References
- No required textbook
- Some suggested references
Some Course References

- No required textbook.
- Put in pictures of the books
- Additional ones for Ocaml given separately

Course Grading

- Homework 10%
- About 12 MPs (in Ocaml) and 12 written assignments
- Submitted by svn
  - MPs – plain text code that compiles; HWs – pdf
- Late submission penalty: 20% of assignments total value
- 2 Midterms - 25% each
  - In class – Oct 7, Nov 11
- DO NOT MISS EXAM DATES!
- Final 40% - Dec 12, 1:30pm – 4:30pm
- Percentages are approximate

Course Homework – Handwritten & 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

Programming Languages & Compilers

Three Main Topics of the Course

I: New Programming Paradigm
II: Language Translation
III: Language Semantics

Order of Evaluation

Specification to Implementation

I : New Programming Paradigm
II: Functional Programming
   Environments and Closures
III: Patterns of Recursion
Continuation Passing Style
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://caml.inria.fr/index.en.html
- To install OCAML on your computer see: http://caml.inria.fr/ocaml/release.en.html

References for OCaml

Supplemental texts (not required):
- The Objective Caml system release 4.0, 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, 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

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)
- Used at Microsoft for writing SLAM and other a formal methods tool for C programs
  - Microsoft variant: F#

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
For the OCAML code for today’s lecture see 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 *)
Characters 0-3:
"Hi there";;  (* has type string *)
Error: This expression has type float but an expression was expected of type string
# 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

# 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 => 3  name => "Steve"  
  y => 17  region => (5.4, 3.7)  
  b => true  id => {Name = "Paul",  
                   Age = 23,  
                   SSN = 999888777}

Global Variable Creation

# 2 + 3;; (* Expression *)  
// doesn't affect the environment  
# let test = 3 < 2;; (* Declaration *)  
val test : bool = false  
// ρ₁ = {test → false}  
# let a = 1 let b = a + 4;; (* Seq of dec *)  
// ρ₂ = {b → 5, a → 1, test → false}

Environments

test => true  
a => 1  b => 5

New Bindings Hide Old

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

What is the environment after this declaration?

Environments

test => 3.7  
a => 1  b => 5

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}
Now it’s your turn

You should be able to do HW1
Problem 1, parts (* 1 *) and (* 2 *)
Booleans (aka Truth Values)

# true;;
- : bool = true
# false;;
- : bool = false
// ρ7 = {c → 4, test → 3.7, a → 1, b → 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
# not (4 > 6);;
- : bool = true

Now it’s your turn
You should be able to do HW1 Problem 1, part (* 5 *)

Tuples as Values

// ρ7 = {c → 4, test → 3.7, a → 1, b → 5} 
# let s = (5, "hi", 3.2);;
val s : int * string * float = (5, "hi", 3.2)

Pattern Matching with Tuples

// ρ8 = {s → (5, "hi", 3.2), 
c → 4, test → 3.7, 
a → 1, b → 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 HW1 Problem 1, part (* 6 *)

Functions

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

Nameless Functions (aka Lambda Terms)

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

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)

First definition syntactic sugar for second
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?

```
- : int = 15
```

Values fixed at declaration time

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

What is the result this time?

```
- : 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:
  \[ \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \]
- Closure for plus_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 *)

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

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

```ocaml
# 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")
```

Match Expressions

```ocaml
# 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>
```

Closure for plus_pair

```ocaml
n
Assume ρplus_pair was the environment just before plus_pair defined

n
Closure for plus_pair:
  <(n,m) → n + m, ρplus_pair>

n
Environment just after plus_pair defined:
  {plus_pair → <(n,m) → n + m, ρplus_pair >} 
  + ρplus_pair
```

Evaluation of Application with Closures

- In environment ρ, evaluate left term to closure, c = <(x1,...,xn) → b, ρ>
- (x1,...,xn) variables in (first) argument
- Evaluate the right term to values, (v1,...,vn)
- Update the environment ρ to
  ρ' = {x1 → v1,..., xn →vn}+ ρ
- Evaluate body b in environment ρ'

Evaluation of Application of plus_pair

- Assume environment
  ρ = {x → 3..., 
       plus_pair →<(n,m) →n + m, ρplus_pair> 
  ρplus_pair}
- Eval (plus_pair (4,x), ρ) =
  Eval (App <(n,m) →n + m, ρplus_pair> (4,x), ρ)) =
  Eval (App <(n,m) →n + m, ρplus_pair> (4,3), ρ)) =
  Eval (n + m, {n -> 4, m -> 3} + ρplus_pair) =
  Eval (4 + 3, {n -> 4, m -> 3} + ρplus_pair) = 7
Closure question

If we start in an empty environment, and we execute:

```plaintext
let f = fun => n + 5;;
(* 0 *)
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
```

What is the environment at (* 0 *?)

---

**Answer**

ρ₀ = {f → <n → n + 5, { }>}

---

Closure question

If we start in an empty environment, and we execute:

```plaintext
let f = fun => n + 5;;
(* 1 *)
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
```

What is the environment at (* 1 *?)

---

**Answer**

ρ₀ = {f → <n → n + 5, { }>}
ρ₁ = {pair_map → <g (n,m) = (g n, g m),
       {f → <n → n + 5, { }>}>},
     f → <n → n + 5, { }>}

---

Closure question

If we start in an empty environment, and we execute:

```plaintext
let f = fun => n + 5;;
(* 2*)
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
```

What is the environment at (* 2 *?)

---

**Answer**
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**

---

Scoping Question

Consider this code:

```ocaml
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