Contact Information – Sasa Misailovic
- Office: 4110 SC
- Office hours:
  - Tuesday, Thursday 12:40pm – 1:30pm
  - Also by appointment
- Email: misailo@illinois.edu

Contact Information - TAs
- Teaching Assistants Office: 0207 SC
- John Lee
  - Email: jlee170@illinois.edu
  - Hours: Tue 9:00am – 9:50am
    Fri 2:00pm – 2:50pm
- Stephen Skeirik
  - Email: skeirik2@illinois.edu
  - Hours: Mon, Thu 10:00am – 10:50am

Course TAs
- John Lee
- Stephen Skeirik

Course Website
- https://courses.engr.illinois.edu/cs421/fa2017/CS421A
- 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

Some Course References
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 Assignments (WA) (~7%)
  - About 6 MPs (in Ocaml) (~7%)
  - About 5 Labs (~6%)
  - All WAs and MPs Submitted by PrairieLearn
- Late submission penalty: 20%
- Labs in Computer-Based Testing Center (Grainger)
- Self-scheduled over a three day period
- No extensions beyond the three day period
- Fall back: Labs become MPs

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.

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
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 4.05, 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)
- Industrially Relevant: Jane Street trades billions of dollars per day using OCaml programs
- Similar languages: Microsoft F#, SML, Haskell, Scala
Session in OCAML

```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

```ocaml
# 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

```ocaml
# 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

```ocaml
# "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**
  
  \[
  \rho = \{\text{name}_1 \rightarrow \text{value}_1, \text{name}_2 \rightarrow \text{value}_2, \ldots\}\n  \]
  
  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 true  

Global Variable Creation

# 2 + 3;; (* Expression *)  
// doesn't affect 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 *)  
// \( \rho_2 = \{b \rightarrow 5, a \rightarrow 1, test \rightarrow false\} \)  

New Bindings Hide Old

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

let test = 3.7;;  

What is the environment after this declaration?

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

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

9/12/17 37

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

Now it’s your turn

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

9/12/17 42
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
# not (4 > 6);;
- : bool = true

Now it's your turn

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

Tuples as Values

// $\rho_8 = \{s \rightarrow (5, "hi", 3.2), c \rightarrow 4, \text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

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)

- : bool = true
# not (4 > 6);;
- : bool = true

9/12/17 43

9/12/17 44

9/12/17 45

9/12/17 46

9/12/17 47

9/12/17 48
Now it’s your turn

You should be able to do WA1 Problem 1, part (* 6 *)

Functions

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

plus_two 17;;
- : int = 19
```

Nameless Functions (aka Lambda Terms)

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

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

Using a nameless function

```ocaml
# (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
define x = 12
val x : int = 12

define 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
define x = 7
val x : int = 7

define 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:
  \[ f \to <(v_1, \ldots, v_n) \to \text{exp}, \rho_f > \]
- Where \( \rho_f \) is the environment in effect when \( f \) is defined (if \( f \) is a simple function)

9/12/17

Closure for plus_x

- When plus_x was defined, had environment:
  \[ \rho_{\text{plus}_x} = \{x \to 12, \ldots\} \]
- Recall: let \( \text{plus}_x y = y + x \)
  is really let \( \text{plus}_x = \text{fun} y \to y + x \)
- Closure for \( \text{fun} y \to y + x \):
  \[ <y \to y + x, \rho_{\text{plus}_x} > \]
- Environment just after plus_x defined:
  \[ \{\text{plus}_x \to <y \to y + x, \rho_{\text{plus}_x} >\} + \rho_{\text{plus}_x} \]

9/12/17

Now it’s your turn

You should be able to do WA1
Problem 1, parts (* 7 *) and (* 8 *)

9/12/17

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

9/12/17

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

9/12/17
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

- Assume $\rho_{plus\_pair}$ was the environment just before `plus_pair` defined
- Closure for `plus_pair`:
  $\langle(n,m) \rightarrow n + m, \rho_{plus\_pair} \rangle$
- Environment just after `plus_pair` defined:
  $\{plus\_pair \rightarrow \langle(n,m) \rightarrow n + m, \rho_{plus\_pair} \rangle\}$ $\ + \rho_{plus\_pair}$