Three Main Topics of the Course

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

I: New Programming Paradigm
- Functional Programming
- Environments and Closures
- Patterns of Recursion
- Continuation Passing Style

II: Language Translation
- Lexing and Parsing
- Type Systems
- Interpretation
Lexing and Parsing
Type Systems
Interpretation
Order of Evaluation
Specification to Implementation

Operational Semantics
Lambda Calculus
Axiomatic Semantics

CS422
CS426
CS477

Office: 2112 SC
Office hours:
- Monday 10:30am – 11:20pm
- Wednesday 1:30pm – 2:20pm
- Also by appointment
Email: egunter@illinois.edu

Teaching Assistants Office: 0207 SC
Paul M Krogmeier
- Email: paulmk2@illinois.edu
- Hours: Wed 2:30pm – 3:20pm
  Fri 2:30pm – 3:20pm
Jacob Scott Laurel
- Email: jlaurel2@illinois.edu
- Hours: Fri 10:00am – 11:40pm
Contact Information - TAs

- Teaching Assistants Office: 0207 SC
  - John J Lee
    - Email: llee170@illinois.edu
    - Hours: Tues 2:00pm – 2:50pm
      Thurs 2:00pm – 2:50pm
  - Liyi Li
    - Email: jlaurel2@illinois.edu
    - Hours: Mon & Fri 1:00pm – 1:50pm

Contact Information – TAs cont

- Leon Ken Medvinsky
  - Email: leonkm2@illinois.edu
  - Hours: Mon 2:30pm – 3:20pm,
      Tues 11:00am-11:50am
- Adithya Murali
  - Email: adithya5@illinois.edu
  - Hours: Tues & Thurs 10:00am – 10:50am

Course Website

- https://courses.engr.illinois.edu/cs421/fa2019
- 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.
- Additional ones for Ocaml given separately

Course Grading

- Assignments 20%
  - About 12 Web Assignments (WA) (~7%)
  - About 5 MPs (in Ocaml) (~6%)
  - About 6 Labs (~7%)
  - All WAs and MPs Submitted by PrairieLearn
  - Late submission penalty: 20%
  - Labs in Computer-Based Testing Center (Grainger)
  - Self-scheduled over a four day period
  - Rules of CBTF apply
  - Fall back: Labs become MPs
Course Grading

- 2 Midterms - 20% each
  - Labs in Computer-Based Testing Center (Grainger)
  - Self-scheduled over a four day period
  - Fall back: In class backup dates – Oct 7, Nov 18
  - BE AVAILABLE FOR FALL BACK DATES!
- Final 40% - CBTF
  - Fall back: In class backup date: Dec 20, 7:00pm-10:00pm
  - Percentages are approximate

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
- Do not have to cite course notes or me

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
  - To try on the web: https://try.ocamlpro.com

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

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
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
  - Major language supported at Bloomberg
- 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 0-3:
  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: \( \rho = \{\text{name}_1 \rightarrow \text{value}_1, \text{name}_2 \rightarrow \text{value}_2, \ldots\} \)
- Often stored as list, or stack.
- To find value, start from left and take first match.

Environments

<table>
<thead>
<tr>
<th>X \rightarrow</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>&quot;Steve&quot;</td>
</tr>
<tr>
<td>y \rightarrow</td>
<td>17</td>
</tr>
<tr>
<td>region</td>
<td>(5.4, 3.7)</td>
</tr>
<tr>
<td>b \rightarrow</td>
<td>true</td>
</tr>
<tr>
<td>id</td>
<td>{Name = &quot;Paul&quot;, Age = 23, SSN = 999888777}</td>
</tr>
</tbody>
</table>

Global Variable Creation

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

New Bindings Hide Old

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

// $\rho_3 = \{\text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$
# let b = 5 * 4
// $\rho_4 = \{b \rightarrow 20, \text{test} \rightarrow 3.7, a \rightarrow 1\}$
in 2 * b;;
- : int = 40
// $\rho_5 = \rho_3 = \{\text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$
# 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
// $\rho_6 = \{b \rightarrow 2\} + \rho_3$
// $\rho_7 = \{\text{test} \rightarrow 3.7, a \rightarrow 1\}$
in b * b;;
val c : int = 4
// $\rho_8 = \{c \rightarrow 4\}$
# 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

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
# 3 > 1 || (print_string "Bye\n"; 4 > 6));;
- : bool = true
# not (4 > 6);;
- : bool = true

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)

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

Nested Tuples

# (*Tuples can be nested *)
let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float = 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 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
Functions

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

First definition syntactic sugar for second

```ocaml
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16
```

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
# 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?

```ocaml
# let x = 7;;  (* New declaration, not an update *)
val x : int = 7
# let plus_x y = y + x;;
val plus_x : int -> int = <fun>
# plus_x 3;;
```

What is the result this time?
Values fixed at declaration time

```ocaml
# 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:
  \[
  \rho_{plus_x} = \{ \ldots, x \rightarrow 12, \ldots \}
  \]
- Recall: let plus_x y = y + x is really let plus_x = fun y -> y + x
- Closure for \( \text{fun } y \rightarrow y + x: \)
  \[
  < y \rightarrow y + x, \rho_{plus_x} >
  \]
- Environment just after plus_x defined:
  \[
  \{ plus_x \rightarrow < y \rightarrow y + x, \rho_{plus_x} > \} + \rho_{plus_x}
  \]

Evaluation of Application of plus_x;;

- Have environment:
  \[
  \rho = \{ plus_x \rightarrow < y \rightarrow y + x, \rho_{plus_x} >, \ldots, y \rightarrow 3, \ldots \}
  \]
  where \( \rho_{plus_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)
- \( \text{Eval } (plus_x y, \rho) \) rewrites to
- \( \text{App } (\text{Eval}(plus_x, \rho), \text{Eval}(y, \rho)) \) rewrites to
- \( \text{App } (< y \rightarrow y + x, \rho_{plus_x} >, 3) \) rewrites to
- \( \text{Eval } (y + x, \{ y \rightarrow 3 \} + \rho_{plus_x}) \) rewrites to
- \( \text{Eval } (3 + 12, \rho_{plus_x}) = 15 \)

Now it’s your turn

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

Partial application of functions

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# 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!
```

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);;
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

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:
  \[(n,m) \rightarrow n + m, \rho_{plus_pair}\]
- Environment just after plus_pair defined:
  \{plus_pair \rightarrow \langle(n,m) \rightarrow n + m, \rho_{plus_pair}\}\]
  + ρ_{plus_pair}