Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

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

Susannah Johnson

Edgar Pek

Yet To Be Determined
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.
Three Main Topics of the Course

I
New Programming Paradigm

II
Language Translation

III
Language Semantics
Programming Languages & Compilers

I
New Programming Paradigm

II
Language Translation

III
Language Semantics

Order of Evaluation

Specification to Implementation
I : New Programming Paradigm

- Functional Programming
- Environments and Closures
- Patterns of Recursion
- Continuation Passing Style
II : Language Translation

Lexing and Parsing

Type Systems

Interpretation
III : Language Semantics

- Operational Semantics
- Lambda Calculus
- Axiomatic Semantics
Programming Languages & Compilers

Order of Evaluation

Operational Semantics
Lambda Calculus
Axiomatic Semantics

Specification to Implementation

CS422
CS426
CS477
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

- OCAML 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

```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

# 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

# let \( x = 2 + 3 ;; \) (* declaration *)
val \( x : \text{int} = 5 \)

# let \( \text{test} = 3 < 2 ;; \)
val \( \text{test} : \text{bool} = \text{false} \)

# let \( a = 1 \) let \( b = a + 4 ;; \) (* Sequence of dec *
val \( a : \text{int} = 1 \)
val \( b : \text{int} = 5 \)
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
X $\rightarrow$ 3

y $\rightarrow$ 17

name $\rightarrow$ “Steve”

region $\rightarrow$ (5.4, 3.7)

b $\rightarrow$ true

id $\rightarrow$ {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
// \(\rho_1 = \{\text{test} \rightarrow \text{false}\}\)
# let a = 1 let b = a + 4;; (* Seq of dec *)
// \(\rho_2 = \{b \rightarrow 5, \ a \rightarrow 1, \ \text{test} \rightarrow \text{false}\}\)
// $\rho_2 = \{b \rightarrow 5, \ a \rightarrow 1, \ \text{test} \rightarrow \text{false}\}$

let test = 3.7;;

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

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

What is the environment after this declaration?

// $\rho_3 = \{\text{test} \rightarrow 3.7, \ a \rightarrow 1, \ b \rightarrow 5\}$
Environments

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 *)
Local Variable Creation

```ocaml
// 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

```ocaml
// 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
```

- `test` is bound to 3.7
- `a` is bound to 1
- `b` is bound to 5
- `c` is bound to $c = a + a * a$
- The result of `b * b` is 25

Initial state:

- `a` → 1
- `test` → 3.7
- `b` → 5

Steps:

1. `b = a + a`
   - `b` → 2
2. `in b * b;;`
   - `c` → 4
   - `b` → 5

Final state:

- `c` → 4
- `b` → 5
- `test` → 3.7
- `a` → 1
- `b` → 5

Notes:

- Local let binding creates a fresh copy of variables within the scope of the binding.
- The `let` expression binds `b` to `a + a`.
- The result of `b * b` is calculated within the binding scope.
- The updated state after the binding is reflected in the final `b` value.

```ocaml
8/28/14  
```
let c =
    let b = a + a
    in b * b;;
val c : int = 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

8/28/14
Now it’s your turn

You should be able to do HW1
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 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)

// 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

# 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

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

plus_two 17;;
- : int = 19
Nameless Functions (aka Lambda Terms)

```
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

```
# 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

```ocaml
# 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;;
val - : 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
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:
  \[ \rho_{plus_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \]
- Closure for plus_x:
  \[ \langle y \rightarrow y + x, \rho_{plus_x} \rangle \]
- Environment just after plus_x defined:
  \[ \{ plus_x \rightarrow \langle y \rightarrow y + x, \rho_{plus_x} \rangle \} + \rho_{plus_x} \]
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 \( \rho_{\text{plus}_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)

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

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

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

- Eval \((3 + 12, \rho_{\text{plus}_x}) = 15\)
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

# 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;;
# 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

```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

# 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 $\rho_{\text{plus\_pair}}$ 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:

  $$\{\text{plus\_pair} \rightarrow <(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} >\} + \rho_{\text{plus\_pair}}$$
Evaluation of Application with Closures

- In environment $\rho$, evaluate left term to closure, $c = \langle (x_1,\ldots,x_n) \rightarrow b, \rho \rangle$
- $(x_1,\ldots,x_n)$ variables in (first) argument
- Evaluate the right term to values, $(v_1,\ldots,v_n)$
- Update the environment $\rho$ to $\rho' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho$
- Evaluate body $b$ in environment $\rho'$
Evaluation of Application of plus_pair

- Assume environment

\[ \rho = \{x \rightarrow 3, ..., \} \]

plus_pair \(\rightarrow\) \(\langle(n,m) \rightarrow n + m, \rho_{\text{plus_pair}}\rangle\} + \rho_{\text{plus_pair}} \)

- Eval (plus_pair (4,x), \(\rho\))=

- Eval (App \(\langle(n,m) \rightarrow n + m, \rho_{\text{plus_pair}}\rangle\) (4,x), \(\rho\)) =

- Eval (App \(\langle(n,m) \rightarrow n + m, \rho_{\text{plus_pair}}\rangle\) (4,3), \(\rho\)) =

- Eval (n + m, \(\{n \rightarrow 4, m \rightarrow 3\}\) + \(\rho_{\text{plus_pair}}\)) =

- Eval (4 + 3, \(\{n \rightarrow 4, m \rightarrow 3\}\) + \(\rho_{\text{plus_pair}}\)) = 7
Closure question

- If we start in an empty environment, and we execute:
  
  ```ml
  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 *?)?
\[ \rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \]
Closure question

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

let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;

(* 1 *)
let f = pair_map f;;

What is the environment at (* 1 *)?
\[ \rho_0 = \{f \rightarrow <n \rightarrow n + 5, \{ \} >\} \]

\[ \rho_1 = \{ \text{pair}_\text{map} \rightarrow <g (n,m) = (g n, g m), \}
\{f \rightarrow <n \rightarrow n + 5, \{ \} >\} >\},
\]

\[ f \rightarrow <n \rightarrow n + 5, \{ \} >\]
Closure question

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

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

(* 2*)

What is the environment at (* 2 *)?
Curried vs Uncurried

- Recall
  
  `val add_three : int -> int -> int -> int = <fun>`

- How does it differ from

  ```ocaml
  # 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*
Curried vs Uncurried

# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
    ^^^^^^^^^^^^^

This function is applied to too many arguments, maybe you forgot a `;'
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
Consider this code:

```ml
let x = 27;;
let f x =
    let x = 5 in
    (fun x -> print_int x) 10;;
f 12;;
```

What value is printed?

1. 5
2. 10
3. 12
4. 27