Programming Languages and Compilers (CS 421)

Sasa Misailovic
4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

Based in part on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter
Contact Information – Sasa Misailovic

- Office: 4110 SC
- Office hours:
  - Tuesday, Thursday 8:30am – 9:30am
  - Also by appointment
- Email: misailo@illinois.edu
Course Website

- [https://courses.engr.illinois.edu/cs421/fa2018/CS421A](https://courses.engr.illinois.edu/cs421/fa2018/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

![Book Covers]

- "Essentials of Programming Languages"
- "Purely Functional Data Structures"
- "Modern Compiler Implementation in ML"
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 through 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 Grading

- 2 Midterms - 20% each
  - Labs in Computer-Based Testing Center (Grainger)
  - Self-scheduled over a three day period
  - No extensions beyond the three day period
  - Dates: Oct 2-4 (Midterm 1) Nov 6-8 (Midterm 2)
  - Fall back: In class backup dates – Oct 9, Nov 13

  **DO NOT MISS EXAM DATES!**

- Final 40% - Dec 19, 8:00am – 11:00am (nominally)
- Will likely use CBTF for Final (3 day window)
- 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.
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
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

8/30/2018
I : New Programming Paradigm

- Functional Programming
- Environments and Closures
- Patterns of Recursion
- Continuation Passing Style
Order of Evaluation

Functional Programming and Recursion
Continuation Passing Style

Specification to Implementation
Programming Languages & Compilers

II : Language Translation

Lexing and Parsing

Type Systems

Interpretation
Programming Languages & Compilers

Order of Evaluation

Lexing

Parsing

Systems

Specification to Implementation

Specification to Implementation
III : Language Semantics

Operational Semantics

Lambda Calculus

Axiomatic Semantics
Programming Languages & Compilers

Order of Evaluation

Specification to Implementation

Operational Semantics

Lambda Calculus

Axiomatic Semantics

CS422

CS426

CS477

8/30/2018
OCAML

- Locally:
  - Compiler is on the EWS-linux systems at /usr/local/bin/ocaml
  - Be sure to *module load ocaml/2.07.0* in EWS!

- Globally:
  - Main CAML home: [http://ocaml.org](http://ocaml.org)
  - To install OCAML on your computer see: [http://ocaml.org/docs/install.html](http://ocaml.org/docs/install.html)
  - Or use one of the online OCAML compilers…
References for OCaml

- Supplemental texts (not required):
- The Objective Caml system release 4.07, 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
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 (e.g. parsing, compilers, user interfaces)
Why Learn OCAML?

- Industrially Relevant: Jane Street trades billions of dollars per day using OCaml programs
- Similar languages: Microsoft F#, SML, Haskell, Scala, Scheme
- Who uses functional programming?
  - Google – MapReduce
  - Microsoft – LinQ
  - Twitter – Scala
  - Bonus: who likes set comprehensions in Python?

```
>>> squares = [x**2 for x in range(10)]
```
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 (back in 1970s)
  - 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
Session in OCAML

% ocaml

Objective Caml version 4.07

# _

# (* 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;;
    ^^^
Error: This expression has type float but an expression was expected of type int

# 1.0 *. 2;; (* No Implicit Coercion *)
Characters 7-8:
  1.0 *. 2;;
    ^^
Error: This expression has type int but an expression was expected of type float
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

```ml
# 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.

- Implementation: 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)

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
// ρ₁ = {test → false}
# let a = 1 let b = a + 4;;  (* Seq of dec *)
// ρ₂ = {b → 5, a → 1, test → false}
New Bindings Hide Old

// $\rho_2 = \{b \to 5, a \to 1, \text{test} \to \text{false}\}$
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?

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

\[
\begin{align*}
test & \rightarrow 3.7 \\
a & \rightarrow 1 \\
b & \rightarrow 5
\end{align*}
\]
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
    in b * b;;

# b;;
Local let binding

// $\rho_5 = \{\text{test} \rightarrow 3.7, \ a \rightarrow 1, \ b \rightarrow 5\}$

# let c =
    let b = a + a

// $\rho_6 = \{b \rightarrow 2\} + \rho_5$
// $= \{b \rightarrow 2, \ \text{test} \rightarrow 3.7, \ a \rightarrow 1\}$
   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

// $\rho_5 = \{\text{test} \rightarrow 3.7, \text{a} \rightarrow 1, \text{b} \rightarrow 5\}$

# let c =
    let b = a + a

// $\rho_6 = \{\text{b} \rightarrow 2\} + \rho_5$
// = \{\text{b} \rightarrow 2, \text{test} \rightarrow 3.7, \text{a} \rightarrow 1\}

in b * b;;

val c : int = 4

// $\rho_7 = \{\text{c} \rightarrow 4, \text{test} \rightarrow 3.7, \text{a} \rightarrow 1, \text{b} \rightarrow 5\}$

# b;;

- : int = 5
Local let binding

// ρ₅ = {test → 3.7, a → 1, b → 5}
# let c =
  let b = a + a
// ρ₆ = {b → 2} + ρ₅
// = {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

// \( \rho_7 = \{c \rightarrow 4, \text{test} \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\} \)
# 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

# not (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
Tuples as Values

// \( \rho_0 = \{ c \rightarrow 4, a \rightarrow 1, b \rightarrow 5 \} \)

# let s = (5, "hi", 3.2);;

val s : int * string * float = (5, "hi", 3.2)

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

// \( \rho = \{ s \rightarrow (5, "hi", 3.2), a \rightarrow 1, b \rightarrow 5, c \rightarrow 4 \} \)

# 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 (a, _, _) = s;;
val a : int = 5

# 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"
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)

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

(* An application *)
# (fun x -> x * 3) 5;;
: int = 15

(* As data *)
# ((fun y -> y +. 2.0), (fun z -> z * 3));;
- : (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

# 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

```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;;

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

```ml
# 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**
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:

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

Where \( \rho \) is the environment in effect when the function is defined (for a simple function).
Recall: let plus_x = fun x => y + x

let x = 12

let plus_x = fun y -&gt; y + x

let x = 7
Closure for plus_x

- When plus_x was defined, had environment:
  \[ \rho_{\text{plus}_x} = \{ \ldots, x \mapsto 12, \ldots \} \]

- Recall: let plus_x y = y + x
  is really let plus_x = fun y -> y + x

- Closure for fun y -> y + x:
  \[ <y \mapsto y + x, \rho_{\text{plus}_x}> \]

- Environment just after plus_x defined:
  \[ \{ \text{plus}_x \mapsto <y \mapsto y + x, \rho_{\text{plus}_x}> \} + \rho_{\text{plus}_x} \]
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
Functions on tuples

# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>

# plus_pair (3,4);;
- : int = 7

# let twice x = (x,x);;
val twice : 'a -> 'a * 'a = <fun>

# twice 3;;
- : int * int = (3, 3)

# twice "hi";;
- : string * string = ("hi", "hi")
Curried vs Uncurried

- Recall

```ocaml
# let add_three u v w = u + v + w;;
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_three 6 3 2;;
- : int = 11

# 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>
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
```

Partial application also called sectioning
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>