Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2021

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

Environments and Closures

Patterns of Recursion

Continuation Passing Style

Order of Evaluation

Specification to Implementation
II : Language Translation

- Lexing and Parsing
- Type Systems
- Interpretation
III : Language Semantics

- Operational Semantics
- Lambda Calculus
- Axiomatic Semantics
Contact Information - Elsa L Gunter

- Office: 2112 SC, also Zoom
- Office hours:
  - Thursday 10:30am – 11:20am
  - Thursday 3:45pm – 2:20pm
  - Also by appointment
- Email: egunter@illinois.edu
Course TAs

Paul Krogmeier  John Lee  Dan Plyukhin

Luhao Wang  Haoqing Zhu
Course Website

- https://courses.engr.illinois.edu/cs421/fa2022
- 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

- No required textbook.
- Pictures of the books on previous slide
- Additional ones for Ocaml given separately
Course Grading

- Assignments 10%
  - Web Assignments (WA) (~5%)
  - MPs (in Ocaml) (5~%)
  - All WAs and MPs Submitted by **PrairieLearn**
  - Late submission penalty: 20% to total
Course Grading

- 2 Midterms - 25% each
  - Sep 29, Nov 10
  - BE AVAILABLE FOR THESE DATES!
- Final 40%
- Fall back: 7:00pm-10:00pm., Tuesday Dec. 13
- 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
OCAML

- Locally:
  - Will use ocaml inside VSCode inside PrairieLearn problems this semester

- 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
  - More notes on this later
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
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)
- 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

Objective Caml version 4.07.1

# (* Read-eval-print loop; expressions and declarations *)

2 + 3;;  (* Expression *)
- : int = 5

# 3 < 2;;

- : bool = false
No Overloading for Basic Arithmetic Operations

```plaintext
# 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 : 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
Booleans (aka Truth Values)

# true;;
- : bool = true

# false;;
- : bool = false

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

\[
\text{fun } n \rightarrow n + 2;;
\]

\[
(f\text{un } n \rightarrow n + 2) 17;;
\]

\[
- : \text{int} = 19
\]
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
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
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)
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

name $\rightarrow$ “Steve”

y $\rightarrow$ 17

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
// 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

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

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

- **test**: 3.7
- **a**: 1
- **b**: 5
Now it’s your turn

You should be able to do WA1-IC Problem 1, parts (* 1 *) - (* 3 * )
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 = ρ₃ = {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
```
let c =
    let b = a + a
    in b * b;;
val c : int = 4

let b =
    let b = a + a
    in b * b;;
b : int = 5
Local let binding

// ρ₅ = ρ₃ = {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
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?
Values fixed at declaration time

```agda
# 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

```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

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

# plus_x 3;;
- : int = 15
```
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:

$$\rho_{\text{plus}_x} = \{\ldots, x \to 12, \ldots\}$$

Recall: \texttt{let plus}_x y = y + x

is really \texttt{let plus}_x = \texttt{fun y -> y + x}

Closure for \texttt{fun y -> y + x}:

$$\langle y \to y + x, \rho_{\text{plus}_x} \rangle$$

Environment just after plus_x defined:

$$\{\text{plus}_x \to \langle y \to y + x, \rho_{\text{plus}_x} \rangle\} + \rho_{\text{plus}_x}$$
Now it’s your turn

You should be able to do WA1-IC Problem 1, parts (* 4 *) - (* 7 *)