Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
https://courses.engr.illinois.edu/cs421/fa2023/CS421D

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

Order of Evaluation

I
New Programming Paradigm

II
Language Translation

III
Language Semantics

Specification to Implementation
Functional Programming
Environments and Closures
Patterns of Recursion
Continuation Passing Style
II : Language Translation

Lexing and Parsing

Type Systems

Interpretation
Order of Evaluation

Lexing and Parsing

Type Systems

Interpretation

Specification to Implementation
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
Contact Information - Elsa L Gunter

- Office: 2112 SC
- Office hours:
  - Tuesday, Thursday 4:00pm – 4:50pm
    - Can attend in zoom
  - Also by appointment
- Email: egunter@illinois.edu
Course TAs

Deeya Bansal
Ben Darnell
Shaurya Gomber
Yerong Li
Alan Yao
James Luo*
Paul Krogmeier*
Course Website

- https://courses.engr.illinois.edu/cs421/fa2023/CS421D
- Main page - summary of news items
- Policy - rules governing course
- Lectures - syllabus and slides
- MPs - information about assignments
- Exams – Syllabi and review material for Midterms and finals
- 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) (~3-6%)
  - MPs (in Ocaml) (~4-7%)
  - All WAs and MPs Submitted by PrairieLearn
  - Late submission penalty: capped at 80% of total
Course Grading

- Five quizzes - 10%
- 3 Midterms - 15% each
  - Sep 14-16, Oct 12-14, Nov 9-11
  - BE AVAILABLE FOR THESE DATES!
- Final 35%
- Tuesday Dec 12, 7:00pm-10:00pm,
- Percentages are approximate
Course Assingments – 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 OCAML 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
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
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
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
```
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
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.
Environments

\[ X \rightarrow 3 \]
\[ y \rightarrow 17 \]
\[ b \rightarrow \text{true} \]

- name \rightarrow “Steve”
- 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}
Environments

- Test $\Rightarrow$ true
- $a \Rightarrow 1$
- $b \Rightarrow 5$
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
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;;
b : int = 5
// 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
Functions

# 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

```ocaml
# 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

# 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

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

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;;
- : 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 formal parameter (the input variables)* with an expression (the function body), written:

  \[
  f \rightarrow < (v_1, ..., v_n) \rightarrow \text{exp}, \ \rho_f >
  \]

- Where \(\rho_f\) is the environment in effect when \(f\) is defined (if \(f\) is a simple function)

* Will come back to the “formal parameter”
Closure for plus_x

- When plus_x was defined, had environment:

\[ \rho_{\text{plus}_x} = \{\ldots, x \rightarrow 12, \ldots\} \]

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

- Closure for fun y -> y + x:

\[ \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle \]

- Environment just after plus_x defined:

\[ \{\text{plus}_x \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle\} + \rho_{\text{plus}_x} \]
Now it’s your turn

You should be able complete ACT1
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 -> int = <fun>
```

Again, first 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>

- What is the value of add_three?
- Let $\rho_{\text{add\_three}}$ be the environment before the declaration
- Remember:

let add_three =
  fun x -> (fun y -> (fun z -> x + y + z));;
Value: $<x \rightarrow \text{fun y} \rightarrow (\text{fun z} \rightarrow x + y + z)$, $\rho_{\text{add\_three}}$ >
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
```
Partial application of functions

```plaintext
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
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!"
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

\[
\rho_8 = \{ s \rightarrow (5, "hi", 3.2), \]
\[
c \rightarrow 4, \ test \rightarrow 3.7, \]
\[
a \rightarrow 1, \ b \rightarrow 5 \}\]

# 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"
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")
```
# 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 plus_pair defined

- Closure for plus_pair:
  
  $\langle(n,m) \rightarrow n + m, \rho_{\text{plus_pair}} \rangle$

- Environment just after plus_pair defined:
  
  $\{\text{plus_pair} \rightarrow \langle(n,m) \rightarrow n + m, \rho_{\text{plus_pair}} \rangle\}$

  $\rho_{\text{plus_pair}}$

Save the Environment!

- A **closure** is a pair of an environment and an association of a pattern (e.g. \((v_1,\ldots,v_n)\) giving the input variables) with an expression (the function body), written:

\[
\langle (v_1,\ldots,v_n) \rightarrow \text{exp}, \rho \rangle
\]

- Where \(\rho\) is the environment in effect when the function is defined (for a simple function)
Evaluating declarations

- Evaluation uses an environment $\rho$
- To evaluate a (simple) declaration $\text{let } x = e$
  - Evaluate expression $e$ in $\rho$ to value $v$
  - Update $\rho$ with $x$ $v$: $\{x \to v\} + \rho$

- Update: $\rho_1 + \rho_2$ has all the bindings in $\rho_1$ and all those in $\rho_2$ that are not rebound in $\rho_1$

$$\{x \to 2, y \to 3, a \to "hi"\} + \{y \to 100, b \to 6\} = \{x \to 2, y \to 3, a \to "hi", b \to 6\}$$
Evaluating expressions in OCaml

- Evaluation uses an environment $\rho$
- A constant evaluates to itself, including primitive operators like + and =
- To evaluate a variable, look it up in $\rho$: $\rho(v)$
- To evaluate a tuple $(e_1, \ldots, e_n)$,
  - Evaluate each $e_i$ to $v_i$, right to left for Ocaml
  - Then make value $(v_1, \ldots, v_n)$
Evaluating expressions in OCaml

- To evaluate uses of +, _, etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: `let x = e1 in e2`
  - Eval `e1` to `v`, then eval `e2` using `{x → v} + ρ`
- To evaluate a conditional expression: `if b then e1 else e2`
  - Evaluate `b` to a value `v`
  - If `v` is `True`, evaluate `e1`
  - If `v` is `False`, evaluate `e2`
Evaluation of Application with Closures

- Given application expression $f \ e$
- In Ocaml, evaluate $e$ to value $v$
- 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
  - $v$ must have form $(v_1,\ldots,v_n)$
- Update the environment $\rho'$ to $\rho'' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
- Evaluate body $b$ in environment $\rho''$