Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
https://courses.engr.illinois.edu/cs421/sp2024

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

Order of Evaluation
Specification to Implementation

Functional Programming
Environments and Closures
Patterns of Recursion
Continuation Passing Style

Lexing and Parsing
Type Systems
Interpretation
Programming Languages & Compilers

Order of Evaluation

Lexing and Parsing
Type Systems
Interpretation

Specification to Implementation

Operational Semantics
Lambda Calculus
Axiomatic Semantics

CS422
CS426
CS477

III : Language Semantics

Contact Information - Elsa L Gunter
- Office: 2112 SC
- Office hours:
  - TBD
  - Can attend in zoom
  - Also by appointment
- Email: egunter@illinois.edu

Course Website
- https://courses.engr.illinois.edu/cs421/sp2024
- 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

![](image1.png) ![](image2.png)

1/15/24
16

Course Grading

- Assignments 10%
  - Web Assignments (WA) (~3-6%)
  - MPs (in Ocaml) (~4-7%)
  - All WAs and MPs Submitted by PrairieLearn
  - Late submission penalty: score capped at 80% of total

1/15/24
18

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 detailed 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 course staff

1/15/24
20

OCAML

- Locally:
  - Will use ocaml inside VSCode inside PrairieLearn problems this semester
- Globally:
  - Main OCAML 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)
  - To try on the web: [https://try.ocamlpro.com](https://try.ocamlpro.com)
  - More notes on this later

1/15/24
21
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

```
# 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
```

```
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

Global Variable Creation

```ocaml
# 2 + 3;; (* Expression *)
// doesn’t affect the environment
# let test = 3 < 2;; (* Declaration *)
val test : bool = false
// \rho_1 = \{ test \rightarrow false \}
# let a = 1 let b = a + 4;; (* Seq of dec *)
// \rho_2 = \{ b \rightarrow 5, a \rightarrow 1, test \rightarrow false \}
```

New Bindings Hide Old

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

What is the environment after this declaration?

```ocaml
// \rho_3 = \{ test \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5 \}
```
Environments

1/15/24

50

Now it’s your turn

You should be able to do WA1-IC Problem 1, parts (* 1 *) - (* 3 *)

Local Variable Creation

// r3 = {test ® 3.7, a ® 1, b ® 5}
# let b = 5 * 4
// r4 = {b ® 20, test ® 3.7, a ® 1}
in 2 * b;;
val c : int = 40
// r5 = r3 = {test ® 3.7, a ® 1, b ® 5}
# b;;
- : int = 5

Local let binding

// r5 = r3 = {test ® 3.7, a ® 1, b ® 5}
# let c =
  let b = a + a
// r6 = {b ® 2} + r3
// r7 = {c ® 4, test ® 3.7, a ® 1, b ® 5}
# b;;
val c : int = 4
// r7 = {c ® 4, test ® 3.7, a ® 1, b ® 5}
# b;;
- : int = 5
Functions

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

plus_two 17;;
- : int = 19
```

Nameless Functions (aka Lambda Terms)

```ocaml
fun n -> n + 2;;

(fun n -> n + 2) 17;;
- : int = 19
```

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?
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

What is the result this time?

What is the result this time?

Question

- Observation: Functions are first-class values in this language
- Question: What value does the environment record for a function variable?
- Answer: a closure
Closure for plus_x

- When plus_x was defined, had environment:
  \( \rho_{\text{plus} \_x} = \{ \ldots, x \to 12, \ldots \} \)
- Recall: let plus_x y = y + x
  is really let plus_x = fun y -> y + x
- Closure for fun y -> y + x:
  \( \langle y \to y + x, \rho_{\text{plus} \_x} \rangle \)
- 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 complete ACT1

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;;

# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16

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
**Functions as arguments**

```ocaml
# 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**

```ocaml
// 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**

```ocaml
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>
```

**Nested Tuples**

```ocaml
(*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)
```

**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**

```ocaml
# 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>
```

---

### Functions as arguments

Let's define a function `thrice` that applies another function `f` three times to an argument `x`:

```ocaml
let thrice f x = f (f (f x));;
```

Then, we define `g` as applying `thrice` to `plus_two` function:

```ocaml
let g = thrice plus_two;;
```

We test `g` on 4:

```ocaml
g 4;;
```

And we apply `thrice` over a function that concatenates strings:

```ocaml
thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
```

### Tuples as Values

We define a tuple `s` as follows:

```ocaml
let s = (5,"hi",3.2);;
```

Then, we define another tuple `d` as a nested tuple:

```ocaml
let d = ((1,4,62),("bye",15),73.95);;
```

### Pattern Matching with Tuples

We define a function `triple_to_pair` that takes a triple and returns a pair:

```ocaml
let triple_to_pair triple = 
  match triple with
  | (0, x, y) -> (x, y)  
  | (x, 0, y) -> (x, y)  
  | (x, y, _) -> (x, y);
```

### Nested Tuples

We define a nested tuple `d`:

```ocaml
let d = ((1,4,62),("bye",15),73.95);;
```

### Functions on tuples

We define a function `plus_pair` that adds two integers:

```ocaml
let plus_pair (n,m) = n + m;;
```

And we define a function `double` that doubles a number:

```ocaml
let double x = (x,x);;
```

### Match Expressions

We define a function `triple_to_pair` using pattern matching:

```ocaml
let triple_to_pair triple =
  match triple
  with (0, x, y) -> (x, y)  
  | (x, 0, y) -> (x, y)  
  | (x, y, _) -> (x, y);;
```
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 $	ext{let } x = e$
  - Evaluate expression $e$ in $\rho$ to value $v$
  - Update $\rho$ with $x : v$: $\{x \rightarrow v\} + \rho$
  - Update: $\rho_1 + \rho_2$ has all the bindings in $\rho_1$ and all those in $\rho_2$ that are not rebound in $\rho_1$
  - $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, b \rightarrow 6\} = \{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

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: $	ext{let } x = e_1 \text{ in } e_2$
  - Eval $e_1$ to $v$, then eval $e_2$ using $\{x \rightarrow v\} + \rho$
- To evaluate a conditional expression:
  if $b$ then $e_1$ else $e_2$
  - Evaluate $b$ to a value $v$
  - If $v$ is True, evaluate $e_1$
  - If $v$ is False, evaluate $e_2$

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 $\rho''$