Three Main Topics of the Course

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

Order of Evaluation
Specification to Implementation
Lexing and Parsing
Type Systems
Interpretation
Specification to Implementation

Order of Evaluation

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

8/25/22 13


Modern Compiler Implementation in ML by Andrew W. Appel, Cambridge University Press 1998

Additional ones for Ocaml given separately

Some Course References

8/25/22 14

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

8/25/22 15

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

8/26/22 16

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

8/25/22 17

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

8/25/22 19
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
% 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

```ocaml
# 15 * 2;;
- : int = 30
# 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

```ocaml
# 1.0 * 2;;  (* No Implicit Coercion *)
## Characters 0-3:
**Error: This expression has type float but an expression was expected of type int**
```

Sequencing Expressions

```ocaml
# "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

```ocaml
# 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)

```ocaml
# true;;
- : bool = true
# false;;
- : bool = false
// \rho_7 = \{c \to 4, test \to 3.7, a \to 1, b \to 5\}
# if b > a then 25 else 0;;
- : int = 25
```

Booleans and Short-Circuit Evaluation

```ocaml
# 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

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

Nameless Functions (aka Lambda Terms)

```ocaml
let plus_two n = n + 2;;
plus_two 17;;
- : int = 19
```
Functions

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

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

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

```
# plus_two 14;;
- : int = 16
```

First definition syntactic sugar for second definition.

Functions with more than one argument

```ocaml
let add_three x y z = x + y + z;;
val add_three : int -> int -> int = <fun>
```

```
let t = add_three 6 3 2;;
val t : int = 11
```

```ocaml
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 definition.

Using a nameless function

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

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

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.

Global Variable Creation

```ocaml
# 2 + 3;; (* Expression *)
// doesn’t affect the environment
```

```ocaml
# let test = 3 < 2;; (* Declaration *)
val test : bool = false
```

```ocaml
// \( \rho_1 = \{ \text{test} \rightarrow \text{false} \}\)
```

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

```
let test = 3.7;;
```

What is the environment after this declaration?

```
// ρ3 = {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 *)
// 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

# 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 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?

X è 12…
X è 7…

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 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:

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

Recall: let plus_x y = y + x

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

Closure for fun y -> y + x:

\[ <y \rightarrow y + x, \rho_{\text{plus}_x}> \]

Environment just after plus_x defined:

\[ \{\text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x}>\} + \rho_{\text{plus}_x} \]

Now it’s your turn

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