Programming Languages and Compilers (CS 421)

Reza Zamani

http://www.cs.illinois.edu/class/cs421/

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

Contact Information – Reza Zamani

- Email: zamani@illinois.edu
- Office hours will be announced on the course webpage.

Contact Information - Choonghwan Lee

- Email: clee83@illinois.edu
- Office hours will be announced on the course webpage.

Course Website

- Main page - summary of news items
- Policy - rules governing course
- Lectures - syllabus and slides
- MPs - information about homework
- Exams
- Unit Projects - for 4 credit students
- Resources - tools and helpful info
- FAQ

Newsgroup

- You **have to** read the newsgroup frequently; it is part of the instruction. Some important clarifications, notifications, ... will appear only there.
- Setup your newsreader to read the news feed "**class.su11.cs421**" at "news.cs.illinois.edu".
- For help to setup your newsreader, see: http://agora.cs.illinois.edu/display/tsg/Newsreader+Documentation

Some Course References

- No required textbook.
- Essentials of Programming Languages (2nd Edition) by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press 2001.
- Compilers: Principles, Techniques, and Tools, (also known as "The Dragon Book"); by Aho, Sethi, and Ullman. Published by Addison-Wesley. ISBN: 0-201-10088-6.
- Modern Compiler Implementation in ML by Andrew W. Appel, Cambridge University Press 1998
- Additional ones for Ocaml given separately

Course Grading

- Homework 25%
  - About 7 MPs (in Ocaml) and maybe 1 written assignment
  - MPs submitted by **handin** on EWS linux machines
  - HWs turned in in class
  - Late submission penalty: 20% of assignments total value
- Midterms - 30%
  - In class –**Jul 14**
- **DO NOT MISS EXAM DATES!**
- Final 45% - time/date will be annouced on the course webpage.

Course Homework

- You may discuss homeworks and their solutions with others
- You may work in groups, but you must list members with whom you worked if you share solutions
- Each student must turn in their own solution separately
- You may look at examples from class and other similar examples from any source
  - Note: University policy on plagiarism still holds - cite your sources if you are not the sole author of your solution
- Problems from homework may appear verbatim, or with some modification on exams

Course Objectives

- New programming paradigm
  - Functional programming
  - Tail Recursion
  - Continuation Passing Style
- Phases of an interpreter / compiler
  - Lexing and parsing
  - Type checking
  - Evaluation
- Programming Language Semantics
  - Lambda Calculus
  - Operational Semantics

OCAML

- Compiler is on the EWS-linux systems at /usr/local/bin/ocaml

WWW Addresses for OCAML

- Main CAML home: http://caml.inria.fr/index.en.html
- To install OCAML on your computer see: http://caml.inria.fr/ocaml/release.en.html

References for CAML

- Supplemental texts (not required):
- The Objective Caml system release 3.09, 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

OCAML

- CAML 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
  - 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

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

- It's fast - winners of the 1999 and 2000 ICFP Programming Contests used OCAML

Scratch Pad

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 (eg parsing, compilers, user interfaces)
- Used at Microsoft for writing SLAM, a formal methods tool for C programs

Session in OCAML

% ocaml

Objective Caml version 3.12.0

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

```
2 + 3;;      (* Expression *)
- : int = 5
# let test = 3 < 2;;       (* Declaration *)
val test : bool = false
```

Session in OCAML

- You may want to use with *ledit*

  % ledit ocaml

  to get better editing capabilities.

Environments

- *Environments* record what value is associated with a given variable
- Central to the semantics and implementation of a language
- Notation

$$\rho = \{\text{name}_1 \to \text{value}_1, \text{name}_2 \to \text{value}_2, \ldots\}$$

Using set notation, but describes a partial function

- Often stored as list, or stack
- To find value start from left and take first match

Sequencing

```
# "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
# let a = 3 let b = a + 2;; (* Sequence of dec *)
val a : int = 3
val b : int = 5
```

Global Variable Creation

2 + 3;; (* Expression *)

// doesn't effect the environment

let test = 3 < 2;; (* Declaration *)

val test : bool = false

// ρ = {test $\rightarrow$ false}

let a = 3 let b = a + 2;; (* Sequence of dec *)

// ρ = {b $\rightarrow$ 5, a $\rightarrow$ 3, test $\rightarrow$ false}

Local let binding

```
# let b = 5 * 4 in 2 * b;;
- : int = 40
//  ρ = {b → 5, a → 3, test → false}
# let c =
    let b = a + a
    in b * b;;
val c : int = 36
//  ρ = {c  → 36, b → 5, a → 3, test → false}
# b;;
- : int = 5
```

Local Variable Creation

```
# let c =
    let b = a + a
//  ρ1 = {b → 5, a → 3, test → false}
 in b * b;;
 val c : int = 36
//  ρ = {c  → 36, b → 5, a → 3, test → false}
# b;;
- : int = 5
```

Terminology

- *Output* refers both to the result returned from a function application
  - As in + outputs integers, whereas +. outputs floats
- Also refers to text printed as a side-effect of a computation
  - As in print_string "\n" outputs a carriage return
  - In terms of values, it outputs ( ) ("unit")
- Typically, we will use "output" to refer to the value returned

No Overloading for Basic Arithmetic Operations

```
# let x = 5 + 7;;
val x : int = 12

# let y = x * 2;;
val y : int = 24

# let z = 1.35 + 0.23;;  (* Wrong type of addition *)
Characters 8-12:
  let z = 1.35 + 0.23;;  (* Wrong type of addition *)
          ^^^^
This expression has type float but is here used with type int
# let z = 1.35 +. 0.23;;
val z : float = 1.58
```

No Implicit Coercion

```
# let u = 1.0 + 2;;
Characters 8-11:
  let u = 1.0 + 2;;
          ^^^
This expression has type float but is here used with
   type int
# let w = y + z;;
Characters 12-13:
  let w = y + z;;
              ^
This expression has type float but is here used with
   type int
```

Booleans (aka Truth Values)

```
# true;;
- : bool = true
# false;;
- : bool = false

# if y > x then 25 else 0;;
- : int = 25
```

Booleans

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

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
```

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
```

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!"
```

Question

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

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:

$$f \to < (v1,\ldots,vn) \to exp, \rho_f >$$

- Where ρ_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:

$$\rho_{plus\_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\}$$

- Closure for plus_x:

$$<y \rightarrow y + x, \rho_{plus\_x} >$$

- Environment just after plus_x defined:

$$\{plus\_x \rightarrow <y \rightarrow y + x, \rho_{plus\_x} >\} + \rho_{plus\_x}$$

Evaluation of Application

- First evaluate the left term to a function (ie starts with fun)
- Evaluate the right term (argument) to a value
  - Things starting with fun are values
- Substitute the argument for the formal parameter in the body of the function
- Evaluate resulting term
- (Need to use environments)

Evaluation Application of plus_x;;

- Have environment:

  $\rho = \{plus\_x \to <y \to y + x, \rho_{plus\_x} >, \ldots ,$
  $y \to 3, \ldots\}$

  where $\rho_{plus\_x} = \{x \to 12, \ldots , y \to 24, \ldots\}$
- Eval $(plus\_x\ y, \rho)$ rewrites to
- Eval $(app <y \to y + x, \rho_{plus\_x} > 3, \rho)$ rewrites to
- Eval $(3 + x, \rho_{plus\_x})$ rewrites to
- Eval $(3 + 12 , \rho_{plus\_x}) = 15$

Scoping Question

Consider this code:

```
let x = 27;;
let f x =
    let x = 5 in
        (fun x -> print_int x) 10;;
f 12;;
```

What value is printed?

5

10

12

27

Recursive Functions

```
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;
  val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
# (* rec  is needed for recursive function
   declarations *)
   (* More on this later *)
```