Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

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

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

Contact Information - Elsa L Gunter

- Office: 2112 SC
- Office hours:
  - Tuesdays 12:30pm – 1:45pm
  - Wednesday 11:00am – 11:50am
  - Thursdays 3:30pm - 4:15pm
  - Also by appointment
- Email: egunter@illinois.edu

Contact Information - TAs

- Teaching Assistants Office: 0207 SC
- Dennis Griffith
  - Email: dgriffi3@illinois.edu
  - Hours: Tues 5:00pm – 5:50pm & Wed 12:30pm – 1:20pm
- Choonghwan Lee
  - Email: clee83@illinois.edu
  - Hours: Mon 9:00am - 9:50am & Fri 1:00pm - 1:50pm

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

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 20%
  - About 9 MPs (in Ocaml) and 3 written assignments
  - MPs submitted by **handin** on EWS linux machines
  - HWs turned in in class
  - Late submission penalty: 20% of assignments total value
- 2 Midterms - 20% each
  - In class – **Oct 11, Nov 10**
- **DO NOT MISS EXAM DATES!**
- Final 40% - Dec 16, 7:00pm – 10:00pm
- Percentages are approximate
  - Exams may weigh more if homework is much better

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 or solution outlines
- 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
- A (possibly better, non-PowerPoint) text version of this lecture can be found at
- http://www.cs.illinois.edu/class/cs421/lectures/ocaml-intro-shell.txt
- For the OCAML code for today's lecture see
- http://www.cs.illinois.edu/class/cs421/lectures/ocaml-intro.ml

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

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
#
# 3 < 2;;
- : bool = false
```

No Overloading for Basic Arithmetic Operations

```
# 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
```

Terminology

- *Output* refers both to the result returned from a function application
  - As in + outputs integers, whereas +. outputs floats
- And 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")
- We will standardly use "output" to refer to the value returned

Declarations; Sequencing of Declarations

```
# let x = 2 + 3;;   (* declaration *)
val x : int = 5
# let test = 3 < 2;;
val test : bool = false
# let a = 3 let b = a + 2;; (* Sequence of dec
  *)
val a : int = 3
val b : int = 5
```

Environments

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

$$\rho = \{name_1 \to value_1, name_2 \to value_2, \ldots\}$$

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

2 + 3;;   (* Expression *)

// doesn't effect the environment

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

val test : bool = false

// ρ = {test → false}

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

// ρ = {b → 5, a → 3, test → 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 → 6, a → 3, test → false}
    in b * b;;
 val c : int = 36
// ρ = {c → 36, b → 5, a → 3, test → false}
# b;;
- : int = 5
```

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
```

Tuples

```
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)
# 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)
```

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

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

Functions on tuples

```
# 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

```
# 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

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
```

Curried vs Uncurried

- Recall

```
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from

```
# let add_triple (u,v,w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is *curried*;
- add_triple is *uncurried*

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!"
```

Curried vs Uncurried

```
# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

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
```

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:

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

- 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:

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

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:

$$<(n,m) \rightarrow n + m, \rho_{plus\_pair}>$$

- Environment just after plus_pair defined:

$$\{plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair} >\} + \rho_{plus\_pair}$$

Evaluation of 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, ρ) rewrites to
- Eval (app $<y \to y + x, \rho_{plus\_x} >$ 3, ρ) rewrites to
- Eval ($y + x$, $\{y \to 3\} + \rho_{plus\_x}$) rewrites to
- Eval ($3 + 12$, ρ_{plus_x}) = 15

Evaluation of Application with Closures

- In environment ρ, evaluate left term to closure, $c = <(x_1,\ldots,x_n) \rightarrow b, \rho>$
- $(x_1,\ldots,x_n)$ variables in (first) argument
- Evaluate the right term to values, $(v_1,\ldots,v_n)$
- Update the environment ρ to

  $\rho' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho$
- Evaluate body b in environment ρ'

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 *)
```