

# Programming Languages and Compilers (CS 421)

---

Elsa L Gunter

2112 SC, UIUC

[http://www.cs.uiuc.edu/class  
/fa06/cs421/](http://www.cs.uiuc.edu/class/fa06/cs421/)

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

# 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.uiuc.edu/class/fa06/cs421/lectures/ocaml-intro-shell.txt>
- For the OCAML code for today's lecture see  
<http://www.cs.uiuc.edu/class/fa06/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 3.08**, by Xavier Leroy, online manual
- **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 3.08.3
```

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

```
# 2 + 3;;    (* Expression *)
```

```
- : int = 5
```

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

```
val test : bool = false
```

# Environments

---

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

- Notation:

$$\rho = \{\text{name}_1 \rightarrow \text{value}_1, \text{name}_2 \rightarrow \text{value}_2, \dots\}$$

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  
//  $\rho = \{\text{test} \rightarrow \text{false}\}$   
# let a = 3 let b = a + 2;; (* Sequence of  
  dec *)  
//  $\rho = \{b \rightarrow 5, a \rightarrow 3, \text{test} \rightarrow \text{false}\}$ 
```

# Local let binding

---

```
# let c =  
  let b = a + a  
  in b * b;;  
val c : int = 36  
  
# b;;  
- : int = 5
```

# Local Variable Creation

---

```
# let c =  
  let b = a + a  
  //  $\rho_1 = \{b \rightarrow 6, a \rightarrow 3, \text{test} \rightarrow \text{false}\}$   
  in b * b;;  
val c : int = 36  
//  $\rho = \{c \rightarrow 36, b \rightarrow 5, a \rightarrow 3, \text{test} \rightarrow \text{false}\}$   
# 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”
- We will standardly 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 or 4 > 6;;
```

```
- : bool = true
```

```
# (print_string "Hi\n"; 3 > 1) or 4 > 6;;
```

```
Hi
```

```
- : bool = true
```

```
# 3 > 1 or (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 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 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:

$$f \rightarrow \langle (v_1, \dots, v_n) \rightarrow \text{exp}, \rho_f \rangle$$

- 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

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

- Closure for plus\_x:

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

# 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 of application of `plus_x;;`

---

Have environment

$$\rho = \{\text{plus\_x} \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus\_x}} \rangle, \dots, \\ y \rightarrow 3, \dots\}$$

where  $\rho_{\text{plus\_x}} = \{x \rightarrow 12, \dots, y \rightarrow 24, \dots\}$

*Eval* (plus\_x y) in  $\rho$  rewrites to

*Eval* (app  $\langle y \rightarrow y + x, \rho_{\text{plus\_x}} \rangle$  3) rewrites to

*Eval* (y + x) in  $(y \rightarrow 3) + \rho_{\text{plus\_x}}$  rewrites to

*Eval* (3 + 12) = 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?

- a) 5
- b) 10
- c) 12
- d) 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 *)
```

# Tuples and Patterns

---

```
# let s = (5,"hi",3.2);;  
val s : int * string * float = (5, "hi", 3.2)  
# let (a,b,c) = s;;  
val a : int = 5  
val b : string = "hi"  
val c : float = 3.2  
# let x = 2, 9.3;; (* tuples don't require parens*)  
val x : int * float = (2, 9.3)
```

# Tuples

---

```
# let d = ((1,4,62),("bye",15),73.95);;  
val d : (int * int * int) * (string * int) * float =  
    ((1, 4, 62), ("bye", 15), 73.95)  
# let (p,(st,_),_) = d;;  
val p : int * int * int = (1, 4, 62)  
val st : string = "bye"
```

# Tuples

---

```
# let fst_of_3 (x,_,_) = x;;  
val fst_of_3 : 'a * 'b * 'c -> 'a = <fun>  
# s;;  
- : int * string * float = (5, "hi", 3.2)  
# fst_of_3 s;;  
- : int = 5  
# fst_of_3 d;;  
- : int * int * int = (1, 4, 62)
```

Notice the type of `fst_of_3`

# 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*

# 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>
```

# Match Expressions

---

```
# let triple_to_pair triple =
```

```
  match triple
```

```
  with (0, x, y) -> (x, y)
```

```
  | (x, 0, y) -> (x, y)
```

```
  | (x, y, 0) -> (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

# Lists

---

- First example of a recursive datatype (aka algebraic datatype)
- Unlike tuples, lists are homogenous in type (all elements same type)

# Lists

---

- List can take one of two forms:
  - Empty list, written  $[]$
  - Non-empty list, written  $x :: xs$ 
    - $x$  is head element,  $xs$  is tail list,  $::$  called “cons”
  - Syntactic sugar:  $[x] == x :: []$   
 $[x1; x2; \dots; xn] == x1 :: x2 :: \dots :: xn :: []$

# Lists

---

```
# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```

# Lists are Homogenous

---

```
# let bad_list = [1; 3.2; 7];;
```

Characters 19-22:

```
let bad_list = [1; 3.2; 7];;
```

^^^

This expression has type float but is here used with type int

# Question

---

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

# Answer

---

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

3 is invalid because of last pair

# Functions Over Lists

---

```
# let rec double_up list =  
  match list  
  with [] -> [] (* pattern before ->,  
expression after *)  
      | (x :: xs) -> (x :: x :: double_up xs);;  
val double_up : 'a list -> 'a list = <fun>  
# let fib5_2 = double_up fib5;;  
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2;  
1; 1; 1; 1]
```

# Functions Over Lists

---

```
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
# let rec poor_rev list =
  match list
  with [] -> []
       | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
# poor_rev silly;;
- : string list = ["there"; "there"; "hi"; "hi"]
```

# Functions Over Lists

---

```
# let rec map f list =  
  match list  
  with [] -> []  
       | (h::t) -> (f h) :: (map f t);;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map plus_two fib5;;  
- : int list = [10; 7; 5; 4; 3; 3]  
# map (fun x -> x - 1) fib6;;  
- : int list = [12; 7; 4; 2; 1; 0; 0]
```

# Iterating over lists

---

```
# let rec fold_left f a list =  
  match list  
  with [] -> a  
       | (x :: xs) -> fold_left f (f a x) xs;;  
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>  
# fold_left  
  (fun () -> print_string)  
  ()  
  ["hi"; "there"];;  
hithere- : unit = ()
```

# Iterating over lists

---

```
# let rec fold_right f list b =  
  match list  
  with [] -> b  
  | (x :: xs) -> f x (fold_right f xs b);;  
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>  
# fold_right  
  (fun s -> fun () -> print_string s)  
  ["hi"; "there"]  
  ();;  
therehi- : unit = ()
```