

# CS421 Topic 2: Introduction to OCaml<sup>1</sup>

Sameer Sundresh  
sundresh@uiuc.edu

University of Illinois at Urbana-Champaign

May 29, 2007

---

<sup>1</sup>Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, Elsa Gunter, and Mark Hills.

## Objectives

### Introducing OCaml

- OCaml Resources

- Some History...

- Features of OCaml

- Why OCaml?

### Writing OCaml Programs

- Running OCaml

- Types

- Environments, Scoping and Functions

- Tuples and Patterns

- Lists

- Other Syntax

- Problem Solutions

# Objectives

Introduction to OCaml: stuff you'll need for MP1

- ▶ how to use the `ocaml` toplevel for interactive programming
- ▶ defining functions
- ▶ basic types: `int`, `float`, `string`
- ▶ compound types: tuples (*pairs*) and lists
- ▶ pattern matching
- ▶ four ways to create variables, and how long they last (*scoping*)
- ▶ *and miscellaneous useful functions along the way. . .*

# Accessing OCaml

- ▶ The OCaml system is installed on the EWS machines
- ▶ Code for the examples run in this lecture will be made available on the lectures page.

# OCaml Online

- ▶ 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>
- ▶ Current release (as of May 16, 2007): 3.10.0

## OCaml References

- ▶ **The Objective Caml system release 3.10**, by Xavier Leroy, online manual
- ▶ **Developing Applications With Objective Caml**, by Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano, published by O'Reilly, available online from course resources

## Some History...

- ▶ 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

## Language Features:

- ▶ higher order functional language
  - ▶ call-by-value parameter passing style
  - ▶ parametric polymorphism (*structural polymorphism*)
  - ▶ automatic garbage collection
  - ▶ user-defined algebraic data types
  - ▶ strongly typed, uses type inference
- 
- ▶ Efficient—the winners of the 2000 and 1999 ICFP Programming Contests used OCaml.

## Why OCaml?

- ▶ Many features not cogently realized in languages you have already learned
- ▶ Assumed basis for much research in programming languages
- ▶ Particularly efficient for programming tasks involving languages (eg parsing, compilers, user interfaces)
- ▶ Compelling real-world applications (SLAM driver verifier at Microsoft, FFTW for optimized FFT generation, Quantitative Investment at Jane St. Capital, etc.)

## Starting the ocaml Interactive Toplevel

- ▶ Interactive toplevel statements must be terminated by ; ;
- ▶ Types are *inferred* if not explicitly provided
- ▶ *Output* can refer to either values returned by expressions or text printed by print calls—we will usually mean the former

```
1 $ ocaml
2     Objective Caml version 3.09.3
3
4 # 34 + 8;;
5 - : int = 42
6 # 27.0 +. 9.4;;
7 - : float = 36.4
8 # "hello";;
9 - : string = "hello"
```

# Hello, World!

- ▶ This is the Standard First Program™
- ▶ The type `unit` is like `void` in C/C++; it represents *commands*.

```
1 # print_string "Hello, world!\n";;  
2 Hello, world!  
3 - : unit = ()
```

- ▶ The type `unit` has one value, `()` (pronounced “unit”).
- ▶ Note:
  - ▶ “Hello, world!” has been output (I/O) to the screen, *but*
  - ▶ the result of this function (the value returned) is `()`.

## Basic Types

```
1 # 20.3;;
2 - : float = 20.3
3 # let answer = 42;;
4 val answer : int = 42
5 # (answer < 50);;
6 - : bool = true
7 # "Ravi";;
8 - : string = "Ravi"
9 #
```

## Type Errors

If you try to combine things with incompatible types, you get a *type error*.

```
1 # 1 + "hello";;
2     ^^^^^^^
3 This expression has type string but is here used
4     with type int
5 # 2.0 +. 4;;
6     ^
7 This expression has type int but is here used
8     with type float
9 #
```

# Environments

*Environments* play a central role in languages.

$$\rho = \{ name_1 \mapsto value_1; name_2 \mapsto value_2; \dots \}$$

- ▶ Mathematically, an environment is a *partial function*
- ▶ Technically a *set*, but can be represented as a *list* (find first match) or a *stack*
- ▶ The environment is usually represented by the letter  $\rho$ .
- ▶ The  $\mapsto$  symbol is pronounced “maps to.”
- ▶ Evaluating an expression always requires an environment
- ▶ Some operations create a new environment from an old one:
  - ▶ global let, local let
  - ▶ function call (*invocation*, not definition!)
  - ▶ match/with

## Variable Creation 1: Global Let

`let name = body ;;`

```

1 # let a=42;;      (* ρ = { a ↦ 42 } *)
2 val a : int = 42
3 # a * 6;;
4 - : int = 252
5 # let i=20;;     (* ρ = { i ↦ 20; a ↦ 42 } *)
6 val i : int = 20
7 # let a=10;;     (* ρ = { a ↦ 10; i ↦ 20 } *)
8 val a : int = 10
9 # a * 2;;
10 - : int = 20
  
```

## Variable Creation 2: Local Variables

$\text{let } name = expression \text{ in } body$

- ▶ This creates  $\rho_2$  from  $\rho_1$ , with `name` having value `expression`.
- ▶  $\rho_2$  is created **after expression is evaluated**, used **only within body**, and then **discarded**.
- ▶ In other words, `name` is **locally scoped**.

```

1 # let i = 10;;      (*  $\rho_1 = \{ i \mapsto 10 \} *$ )
2 val i : int = 10
3 # let a = 20;;     (*  $\rho_1 = \{ a \mapsto 20; i \mapsto 10 \} *$ )
4 val a : int = 20
5 # let i = a + a    (*  $\rho_1 = \{ a \mapsto 20; i \mapsto 10 \} *$ )
6   in i * a;;      (*  $\rho_2 = \{ i \mapsto 40; a \mapsto 20 \} *$ )
7 - : int = 800
8 # i * a;;         (*  $\rho_1 = \{ a \mapsto 20; i \mapsto 10 \} *$ )
9 - : int = 200
  
```

## Variable Creation 3: Function Definition

*let name parameters = body*

```
1 # let i = 10;;  
2 val i : int = 10  
3 # let f n = i + 10;; (* Creates a function *)  
4 val f : 'a -> int = <fun> (* note the arrow! *)  
5 # f i;; (* Function calls are denoted *)  
6 - : int = 20 (* by juxtaposition *)  
7 # let i = 100;;  
8 val i : int = 100  
9 # f i;;  
10 - : int = ... ? (* What should this be? *)
```

## Alternate Syntax

```
1 # let f n = n + 10 ;;  
2 val f : int -> int = <fun>  
3  
4 # let f = fun n -> n + 10 ;;  
5 val f : int -> int = <fun>  
6  
7 # let g x y = x + y ;;  
8 val g : int -> int -> int = <fun>  
9  
10 # let g = fun x y -> x + y ;;  
11 val g : int -> int -> int = <fun>
```

## Save the Environment!

- ▶ A function is a *value* in this language.
- ▶ It is stored internally as a *closure*: code + environment

$$(\lambda v. \text{exp}, \rho_f)$$

- ▶ The code of the function is stored in the first half, the environment *as it existed when the function was created* is stored in the second half as  $\rho_f$ .
- ▶  $\lambda v. \text{exp}$  is read “a function that takes an argument  $v$  and then evaluates  $\text{exp}$ ”—an *anonymous* function.
- ▶ When the function is called (or *applied* to an argument), the variable is instantiated in a *copy* of  $\rho_f$ .

## Function Calls

```

1 # let i = 10;;          (*  $\rho_1 = \{i \mapsto 10\}$  *)
2 val i : int = 10
3 # let f n = i + 10;;   (*  $\rho_2 = \{f \mapsto ((\lambda n. i + 10), \rho_1)\}$  *)
4 val f : 'a -> int = <fun> (*  $\cup \rho_1$  *)
5 # f i;;
6 - : int = 20          (*  $\rho_{1a} = \{n \mapsto 10; i \mapsto 10\}$  *)
7 # let i = 100;;      (*  $\rho_3 = \{i \mapsto 100; f \mapsto ((\lambda n. i + 10), \rho_1)\}$  *)
8 val i : int = 100
9 # f i;;
10 - : int = ... ?     (*  $\rho_{1b} = \{n \mapsto 100; i \mapsto 10\}$  *)
  
```

When `f` is called, we use  $\rho_3$  to look up parameter `i`, and assign that to `n` in a copy of  $\rho_1$ . We use this new environment ( $\rho_{1a}$  or  $\rho_{1b}$ ) to look up the value of `i` inside the function.

# Anonymous Functions

Functions can also be used without first being given names.

```
1 # (fun n -> n + 10) 20 ;;  
2 - : int = 30  
3  
4 # (fun x y -> x + y) 3 4 ;;  
5 - : int = 7
```

## Functions as Parameters

Functions can be passed to other functions – functions which take other functions as parameters are referred to as *higher-order*

```
1 # (fun x y -> x y) (fun x -> x + 5) 4 ;;
2 - : int = 9
3 # let thrice f x = f(f(f x));;
4 val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
5 # let inc n = n + 1 ;;
6 val inc : int -> int = <fun>
7 # thrice inc 4 ;;
8 - : int = 7
9 # let inc3 = thrice inc ;;
10 val inc3 : int -> int = <fun>
11 # inc3 4 ;;
12 - : int = 7
```

## Function Call Steps

- ▶ First evaluate the left term to a function (actually, a *closure*)
- ▶ Evaluate the right term (argument) to a value (remember, functions **are** values!)
- ▶ Substitute the argument for the formal parameter in the body of the function
- ▶ Evaluate resulting term

# Problem 1

1. (106) What will be the output of the following code?

```
1 let x = 20;;  
2 let f = fun x -> x + 1;;  
3 let y = f 30;;  
4 print_int x;;
```

## Function Types

*let name parameters = body*

```
1 # let inc x = x + 1;;
2 val inc : int -> int = <fun>
3 # inc 5;; (* Notice the arrow! *)
4 - : int = 6
5 # inc 2.4;;
6 Toplevel input:
7 # inc 2.4;;
8    ^^^
9 This expression has type float but is here used
10    with type int
```

Parameters exist *only within the function*.

## Functions with Multiple Parameters

Functions are *values* in this language!

```
1 # let triangle a b = a * a + b * b;;  
2 val triangle : int -> int -> int = <fun>  
3 # triangle 3 4;;  
4 - : int = 25  
5 # let t3 = triangle 3;;  
6 val t3 : int -> int = <fun>  
7 # t3 4;;  
8 - : int = 25
```

# Currying

*Currying* is the concept that functions can be written that take just *one* argument at a time, yielding intermediate functions until a final result is returned. Functions that are *uncurried* take all parameters at once (as a tuple).

*Note: named after logician Haskell Curry. Not a culinary reference.*

## Local Let

- ▶ You can create local variables using the `let/in` construct.

```

1 # let triangle a b =
2   let asq = a * a in (*ρ1 = {asq ↦ a * a} ∪ ρf *)
3   let bsq = b * b in (*ρ2 = {bsq ↦ b * b} ∪ ρ1 *)
4     asq + bsq;;
5 val triangle : int -> int -> int = <fun>
  
```

- ▶ The variables `asq` and `bsq` are created after `triangle` is called, once the `let` expressions are reached, and destroyed once the `let` expressions are exited.

```

1 # (let x = 10 in x) + (let x = 20 in x);;
2 - : int = 30 (* How many environments used here? *)
  
```

## Scoping Question

Consider this code:

```
1 let x = 27;;  
2 let foo x =  
3     let x = 5 in  
4         (fun x -> print_int x) 10;;  
5 foo 12;;
```

What value will be printed?

- ▶ a) 5
- ▶ b) 10
- ▶ c) 12
- ▶ d) 27

# Tuples

- ▶ Tuples are one kind of *compound type*.

```

1 # let t = 2,3;;
2 val t : int * int = 2, 3
3 # let t2 = (3,42);;
4 val t2 : int * int = 3, 42
5 # let t3 = (2, "string", true, fun x -> x + x);;
6 val t3 : int * string * bool * (int -> int) =
7   (2, "string", true, <fun>)
8 # let t4 = (1,(2,(3,4)));;
9 val t4 : int * (int * (int * int)) = (1, (2, (3, 4)))
  
```

## Deconstructing Tuples

```
1 # let t3 = (2, "string", true, fun x -> x + x);;  
2 val t3 : int * string * bool * (int -> int) =  
3   (2, "string", true, <fun>)  
4 # let (a,b,c,d) = t3;;  
5 val a : int = 2  
6 val b : string = "string"  
7 val c : bool = true  
8 val d : int -> int = <fun>  
9 # let (n,_,_,f) = t3 in f n ;;  
10 - : int = 4
```

# Pattern Matching

```
1 # let iszero n =  
2     match n with  
3     | 0 -> "it's zero"  
4     | 1 -> "it's one"  
5     | _ -> "not zero or one";;  
6 val iszero : int -> string = <fun>  
7 # iszero 1;;  
8 - : string = "it's one"
```

- ▶ The *patterns* all need to have the same type.
- ▶ Also, the results....

## Alternate Syntax

```
1 # let iszero = function
2   0 -> "it's zero"
3   | 1 -> "it's one"
4   | _ -> "not zero or one";;
5 val iszero : int -> string = <fun>
6
7 # let getfirst = function
8   (a,b,c) -> a ;;
9 val getfirst : 'a * 'b * 'c -> 'a = <fun>
```

## Variable Creation Method 4: Matching

- ▶ Use `match/with` to deconstruct compound types,

```

1 # let inctup a =
2     match a with
3     | (0,y) -> y, 1
4     | (x,y) -> x+1, y+1;;
5 val inctup : int * int -> int * int = <fun>
6 # inctup (2,3);;
7 - : int * int = 3, 4
8 # inctup (0,3);;
9 - : int * int = 3, 1
  
```

Var created with `match` only accessible in first expression after `->`

# Lists

- ▶ A *list* can take two forms:
  - ▶ empty list
  - ▶ an element, followed by the rest of the list
- ▶ Empty lists are written `[]`
- ▶ Non-empty lists are written `x :: xs`
  - ▶  $x$  is the *head* of the list.
  - ▶  $xs$  is the *tail* of the list.
- ▶ Can also write `[x1; x2; ...; xn]`
- ▶ Unlike tuples, lists are *monomorphic*—all elements must have the same type.

## List Examples

```
1 # [];;
2 - : 'a list = []
3 # let empty = [];;
4 val empty : 'a list = []
5 # let single = [1];;
6 val single : int list = [1]
7 # let rlist = [2.3; 4.2; 5.3];;
8 val rlist : float list = [2.3; 4.2; 5.3]
9 # let badlist = [3; 4; 3.14159];;
10 Characters 21-28:
11 This expression has type float but is here used
12     with type int
```

## Problem 2

1. (107) One of the lists below is invalid. Which one?
- ▶ a) [2; 3; 4; 6]
  - ▶ b) [2,3; 4,5; 6,7]
  - ▶ c) [2.3,4; 3.2,5; 6,7.2]
  - ▶ d) [{"hi"; "there"}; ["how"]; []; ["goezit"]]

## Adding to lists

- ▶ `::` prepends element to a list
- ▶ `@` concatenates two lists

```
1 - let l1 = [3;6;9];;  
2 val l1 : int list = [3;6;9]  
3 - let l2 = [4;7;10];;  
4 val l2 : int list = [4;7;10]  
5 - 5 :: l1;;  
6 val - : int list = [5;3;6;9]  
7 - 10 :: 20 :: l2;;  
8 val - : int list = [10;20;4;7;10]  
9 - l1 @ l2;;  
10 val - : int list = [3;6;9;4;7;10]
```

## Pattern Matching with Lists

- ▶ The `match/with` construction works with lists, too.

```
1 # let getfirst l =  
2     match l with  
3     | [] -> 0  
4     | x::xs -> x;;  
5 val getfirst : int list -> int = <fun>  
6 # getfirst [];;  
7 - : int = 0  
8 # getfirst [3;4;5];;  
9 - : int = 3
```

## Polymorphic List Functions

```
1 # let isempty l =  
2     match l with  
3     | [] -> "empty"  
4     | _ -> "nonempty";;  
5 val isempty : 'a list -> string = <fun>  
6 # isempty ["hi";"there"];;  
7 - : string = "nonempty"  
8 # isempty [2;3];;  
9 - : string = "nonempty"
```

## Combining Types

- ▶ Types can be combined arbitrarily

```
1 # let e1 = [ [20;30]; [10;5;2]; [3;6]; [] ];;  
2 val e1 : int list list = [[20; 30]; [10; 5; 2]; ...  
3 # let e2 = [ 2,3; 4,5; 6,7 ];;  
4 val e2 : (int * int) list = [2, 3; 4, 5; 6, 7]
```

# If

- ▶ if can be used as a *command* or an *expression*.
- ▶ then and else branches must have the same type.

```

1 # if (3 < 5) then print_string "hi!\n";;
2 hi!                                (* Command *)
3 - : unit = ()
4 # let x = 10;;
5 val x : int = 10
6 # let y = (if x < 10 then 40 else 50) * 2;;
7 val y : int = 100                    (* Expression *)
8 #
  
```

# Sequencing

```
1 (* Expression sequencing *)
2 # (print_string "Bye\n"; 25);;
3 Bye
4 - : int = 25
5 (* Declaration sequencing *)
6 # let a = 3
7     let b = a + 2;;
8 val a : int = 3
9 val b : int = 5
```

# Problem 1

What will be the output of the following code?

```
1 let x = 20;;  
2 let f = fun x -> x + 1;;  
3 let y = f 30;;  
4 print_int x;;
```

**Answer:** 20

## Problem 2

One of the lists below is invalid. Which one?

- ▶ a) [2; 3; 4; 6]
- ▶ b) [2,3; 4,5; 6,7]
- ▶ c) **Answer** [2.3,4; 3.2,5; 6,7.2]
- ▶ d) [{"hi"; "there"}; {"how"}; []; {"goezit"}]

The first two elements are of type `float * int`, but the last element is of type `int * float`.

---