Graduate Programming Languages: OCaml Tutorial

Dan Grossman
2012

What is this

These slides contain the same code as **`play.ml`** and other files

- Plus some commentary
- Make of them what you will

(Live demos probably work better, but if these slides are useful reading, then great)

This "tutorial" is heavily skewed toward the features we need for studying programming languages

- Plus some other basics

Hello, World!

```
(* our first program *)
let x = print_string "Hello, World!\n"
```

- A *program* is a sequence of *bindings*
- One kind of binding is a *variable binding*
- Evaluation evaluates bindings in order
- To evaluate a variable binding:
  - Evaluate the expression (right of =) in the environment created by the *previous* bindings.
  - This produces a value.
  - Extend the (top-level) environment, binding the variable to the value.

Some variations

```
let x = print_string "Hello, World!\n"
(*same as previous with nothing bound to ()*)
let _ = print_string "Hello, World!\n"
(*same w/ variables and infix concat function*)
let h = "Hello, "
let w = "World!\n"
let _ = print_string (h ^ w)
(*function f: ignores its argument & prints*)
let f x = print_string (h ^ w)
(*so these both print (call is juxtapose)*)
let y1 = f 37
let y2 = f f (* pass function itself *)
(*but this does not (y1 bound to ())*)
let y3 = y1
```


Compiling/running

`ocamlc file.ml`	compile to bytecodes (put in executable)
`ocamlopt file.ml`	compile to native (1-5x faster, no need in class)
`ocamlc -i file.ml`	print types of all top-level bindings (an interface)
`ocaml`	read-eval-print loop (see manual for directives)
`ocamlprof`, `ocamldebug`, …	see the manual (probably unnecessary)

- Later: multiple files

Installing, learning

- Links from the web page:
  - **`www.ocaml.org`**
  - The on-line manual (great reference)
  - An on-line book (less of a reference)
  - Installation/use instructions
- Contact us with install problems soon!
- Ask questions (we know the language, want to share)

Types

- Every expression has one type. So far:

  int string unit t1->t2 'a

```
(* print_string : string->unit, "…" : string *)
let x = print_string "Hello, World!\n"
(* x : unit *)
…
(* ^ : string -> string -> string *)
let f x = print_string (h ^ w)
(* f : 'a -> unit *)
let y1 = f 37 (* y1 : unit *)
let y2 = f f  (* y2 : unit *)
let y3 = y1   (* y3 : unit *)
```


Explicit types

- You (almost) never need to write down types
  - But can help debug or document
  - Can also constrain callers, e.g.:

```
let f x = print_string (h ^ w)
let g (x:int) = f x

let _ = g 37
let _ = g "hi" (*no typecheck, but f "hi" does*)
```


Theory break

Some terminology and pedantry to serve us well:

- Expressions are *evaluated* in an environment
- An *environment* maps variables to values
- Expressions are *type-checked* in a context
- A *context* maps variables to types

- *Values* are integers, strings, function-closures, …
  - "things already evaluated"
- Constructs have evaluation rules (except values) and type-checking rules

Recursion

- A let binding is not in scope for its expression, so:

  let rec

```
(* smallest infinite loop *)
let rec forever x = forever x
(* factorial (if x>=0, parens necessary) *)
let rec fact x =
  if x==0 then 1 else x * (fact(x-1))
(*everything an expression, e.g., if-then-else*)
let fact2 x =
  (if x==0 then 1 else x * (fact(x-1))) * 2 / 2
```


Locals

- Local variables and functions much like top-level ones (with `in` keyword)

```
let quadruple x =
  let double y = y + y in
  let ans = double x + double x in
  ans

let _ =
print_string((string_of_int(quadruple 7)) ^ "\n")
```


Anonymous functions

- Functions need not be bound to names
  - In fact we can *desugar* what we have been doing

```
let quadruple2 x =
  (fun x -> x + x) x + (fun x -> x + x) x

let quadruple3 x =
  let double = fun x -> x + x in
  double x + double x
```

Passing functions

```
(* without sharing (shame) *)
print_string((string_of_int(quadruple  7)) ^ "\n");
print_string((string_of_int(quadruple2 7)) ^ "\n");
print_string((string_of_int(quadruple3 7)) ^ "\n")
(* with "boring" sharing (fine here) *)
let print_i_nl i =
  print_string ((string_of_int i) ^ "\n")
let _ = print_i_nl (quadruple  7);
        print_i_nl (quadruple2 7);
        print_i_nl (quadruple3 7)
(* passing functions instead *)
let print_i_nl2 i f = print_i_nl (f i)
let _ = print_i_nl2 7 quadruple ;
        print_i_nl2 7 quadruple2;
        print_i_nl2 7 quadruple3
```


Multiple args, currying

```
let print_i_nl2 i f = print_i_nl (f i)
```

- Inferior style (fine, but OCaml novice):

```
let print_on_seven f = print_i_nl2 7 f
```

- Partial application (elegant and addictive):

```
let print_on_seven = print_i_nl2 7
```

- Makes no difference to callers:

```
let _ = print_on_seven quadruple ;
        print_on_seven quadruple2;
        print_on_seven quadruple3
```


Currying exposed

```
(* 2 ways to write the same thing *)
let print_i_nl2 i f = print_i_nl (f i)
let print_i_nl2 =
  fun i -> (fun f -> print_i_nl (f i))
(*print_i_nl2 : (int -> ((int -> int) -> unit))
  i.e.,         (int -> (int -> int)  -> unit)
*)

(* 2 ways to write the same thing *)
print_i_nl2 7 quadruple

(print_i_nl2 7) quadruple
```


Elegant generalization

- Partial application is just an *idiom*
  - Every function takes exactly one argument
  - Call (application) "associates to the left"
  - Function types "associate to the right"
- Using functions to simulate multiple arguments is called currying (somebody's name)
- OCaml implementation plays cool tricks so full application is efficient (merges *n* calls into 1)

Closures

Static (a.k.a. lexical) scope; a really big idea

```
let y = 5
let return11 = (* unit -> int *)
   let x = 6 in
   fun () -> x + y
let y = 7
let x = 8
let _ = print_i_nl (return11 ()) (* prints 11! *)
```


The semantics

A function call `e1 e2`:

1. evaluates `e1`, `e2` to values `v1`, `v2` (order undefined) where `v1` is a function with argument `x`, body `e3`
2. Evaluates `e3` in the environment where `v1` was defined, extended to map `x` to `v2`

Equivalent description:

- A function `fun x -> e` evaluates to a triple of `x`, `e`, and the current environment
  - Triple called a *closure*
- Call evaluates closure's body in closure's environment extended to map `x` to `v2`

Closures are closed

```
let y = 5
let return11 = (* unit -> int *)
   let y = 6 in
   fun () -> x + y
```

`return11` is bound to a value `v`

- All you can do with this value is call it (with `()`)
- It will *always* return 11
  - Which environment is not determined by caller
  - The environment contents are immutable
- `let return11 () = 11`
  guaranteed not to change the program

Another example

```
let x = 9
let f () = x+1
let x = x+1
let g () = x+1
let _ = print_i_nl (f() + g())
```


Mutation exists

There is a built-in type for mutable locations that can be read and assigned to:

```
let x = ref 9
let f () = (!x)+1
let _ = x := (!x)+1
let g () = (!x)+1
let _ = print_i_nl (f() + g())
```

While sometimes awkward to avoid, need it much less often than you think (and it leads to sadness)

On homework, do not use mutation unless we say

Summary so far

- Bindings (top-level and local)
- Functions
  - Recursion
  - Currying
  - Closures
- Types
  - “base” types (**unit**, **int**, **string**, **bool**, …)
  - Function types
  - Type variables

Now: compound data

Record types

```
type int_pair = {first : int; second : int}
let sum_int_pr x = x.first + x.second
let pr1 = {first = 3; second = 4}
let _ = sum_int_pr pr1
     + sum_int_pr {first=5;second=6}
```

A type constructor for polymorphic data/code:

```
type 'a pair = {a_first : 'a; a_second : 'a}
let sum_pr f x = f x.a_first + f x.a_second
let pr2 = {a_first = 3; a_second = 4}(*int pair*)
let _ = sum_int_pr pr1
       + sum_pr (fun x->x) {a_first=5;a_second=6}
```


More polymorphic code

```
type 'a pair = {a_first : 'a; a_second : 'a}
let sum_pr f x = f x.first + f x.second
let pr2 = {a_first = 3;    a_second = 4}
let pr3 = {a_first = "hi"; a_second = "mom"}
let pr4 = {a_first = pr2;  a_second = pr2}
let sum_int      = sum_pr (fun x -> x)
let sum_str      = sum_pr String.length
let sum_int_pair = sum_pr sum_int
let _ = print_i_nl (sum_int pr2)
let _ = print_i_nl (sum_str pr3)
let _ = print_i_nl (sum_int_pair pr4)
```

Each-of vs. one-of

- Records build new types via "each of" existing types
- Also need new types via "one of" existing types
  - Subclasses in OOP
  - Enums or unions (with tags) in C
- OCaml does this directly; the tags are *constructors*
  - Type is called a *datatype*

Datatypes

```
type food = Foo of int | Bar of int_pair
          | Baz of int * int | Quux

let foo3     = Foo (1 + 2)
let bar12    = Bar pr1
let baz1_120 = Baz(1,fact 5)
let quux     = Quux (* not much point in this *)

let is_a_foo x =
  match x with (* better than "downcasts" *)
    Foo i     -> true
  | Bar pr    -> false
  | Baz(i,j)  -> false
  | Quux      -> false
```


Datatypes

- Syntax note: Constructors capitalized, variables not
- Use constructor to make a value of the type
- Use pattern-matching to use a value of the type
  - Only way to do it
  - Pattern-matching actually much more powerful

Booleans revealed

Predefined datatype (violating capitalization rules ☹):

```
type bool = true | false
```

`if` is just sugar for `match` (but better style):

- `if e1 then e2 else e3`
-

```
match e1 with
  true  -> e2
| false -> e3
```


Recursive types

A datatype can be recursive, allowing data structures of unbounded size

And it can be polymorphic, just like records

```
type int_tree = Leaf
              | Node of int * int_tree * int_tree
type 'a lst = Null
            | Cons of 'a * 'a lst
let lst1 = Cons(3,Null)
let lst2 = Cons(1,Cons(2,lst1))
(* let lst_bad = Cons("hi",lst2) *)
let lst3 = Cons("hi",Cons("mom",Null))
let lst4 = Cons (Cons (3,Null),
           Cons (Cons (4,Null), Null))
```


Recursive functions

```
type 'a lst = Null
            | Cons of 'a * 'a lst

let rec length lst = (* 'a lst -> int *)
  match lst with
    Null -> 0
  | Cons(x,rest) -> 1 + length rest
```

Recursive functions

```
type 'a lst = Null
            | Cons of 'a * 'a lst

let rec sum lst = (* int lst -> int *)
  match lst with
   Null -> 0
 | Cons(x,rest) -> x + sum rest
```


Recursive functions

```
type 'a lst = Null
            | Cons of 'a * 'a lst

let rec append lst1 lst2 =
(* 'a lst -> 'a lst -> 'a lst *)
  match lst1 with
    Null -> lst2
  | Cons(x,rest) -> Cons(x, append rest lst2)
```


Another built-in

Actually the type `'a list` is built-in:

- Null is written `[]`
- `Cons(x,y)` is written `x::y`
- And sugar for list literals `[5; 6; 7]`

```
let rec append lst1 lst2 = (* built-in infix @ *)
  match lst1 with
    [] -> lst2
  | x::rest -> x :: append rest lst2
```


Summary

- Now we really have it all
  - Recursive higher-order functions
  - Records
  - Recursive datatypes
- Some important odds and ends
  - Tuples
  - Nested patterns
  - Exceptions
- Then (simple) modules

Tuples

Defining record types all the time is unnecessary:

- Types: `t1 * t2 * … * tn`
- Construct tuples `e1,e2,…,en`
- Get elements with pattern-matching `x1,x2,…,xn`
- Advice: use parentheses

```
let x = (3,"hi",(fun x -> x), fun x -> x ^ "ism")

let z = match x with (i,s,f1,f2) -> f1 i

let z = (let (i,s,f1,f2) = x in f1 i)
```


Pattern-matching revealed

- You can pattern-match anything
  - Only way to access datatypes and tuples
  - A variable or _ matches anything
  - Patterns can nest
  - Patterns can include constants (3, “hi”, …)
- `let` can have patterns, just sugar for `match`!
- “Quiz”: What is
  - `let f x y = x + y`
  - `let f pr = (match pr with (x,y) -> x+y)`
  - `let f (x,y) = x + y`
  - `let f (x1,y1) (x2,y2) = x1 + y2`

Fancy patterns example

```
type sign = P | N | Z

let multsign x1 x2 =
  let sign x =
    if x>=0 then (if x=0 then Z else P) else N
  in
  match (sign x1,sign x2) with
      (P,P) -> P
    | (N,N) -> P
    | (Z,_) -> Z
    | (_,Z) -> Z
    | _     -> N (* many say bad style! *)
```

To avoid *overlap*, two more cases
(more robust if datatype changes)

Fancy patterns example

```
exception ZipLengthMismatch

let rec zip3 lst1 lst2 lst3 =
 match (lst1,lst2,lst3) with
     ([],[],[]) -> []
   | (hd1::tl1,hd2::tl2,hd3::tl3) ->
       (hd1,hd2,hd3)::(zip3 tl1 tl2 tl3)
   | _ -> raise ZipLengthMismatch
```

Try that in your favorite language ☺

`'a list -> 'b list -> 'c list -> ('a*'b*'c) list`

Modules

- So far, only way to hide things is local let
  - Not good for large programs
  - OCaml has a great *module system*, but we need only the basics
- Modules and signatures give
  - Namespace management
  - Hiding of values and types
  - Abstraction of types
  - Separate compilation
- By default, OCaml builds on the filesystem

Module pragmatics

- **foo.ml** defines module **Foo**
- **Bar** uses variable **x**, type **t**, constructor **C** in **Foo** via **Foo.x**, **Foo.t**, **Foo.C**
  - Can **open** a module, use sparingly
- **foo.mli** defines signature for module **Foo**
  - Or "everything public" if no **foo.mli**
- Order matters (command-line)
  - No forward references (long story)
  - Program-evaluation order
- See manual for **.cm[i,o]** files, **-c** flag, etc.

Module example

foo.ml

```
type t1 = X1 of int
        | X2 of int

let get_int t =
  match t with
    X1 i -> i
  | X2 i -> i

type even = int

let makeEven i = i*2
let isEven1 i = true
(* isEven2 is "private" *)
let isEven2 i = (i mod 2)=0
```

foo.mli

```
(* choose to show *)
type t1 = X1 of int
        | X2 of int

val get_int : t1->int

(* choose to hide *)
type even

val makeEven : int->even
val isEven1 : even->bool
```


Module example

bar.ml

```
type t1 = X1 of int
        | X2 of int

let conv1 t =
  match t with
    X1 i -> Foo.X1 i
  | X2 i -> Foo.X2 i
let conv2 t =
  match t with
    Foo.X1 i -> X1 i
  | Foo.X2 i -> X2 i

let _ =
 Foo.get_int(conv1(X1 17));
 Foo.isEven1(Foo.makeEven 17)
 (* Foo.isEven1 34 *)
```

foo.mli

```
(* choose to show *)
type t1 = X1 of int
        | X2 of int

val get_int : t1->int

(* choose to hide *)
type even

val makeEven : int->even
val isEven1 : even->bool
```

Not the whole language

- Objects
- Loop forms (bleach)
- Fancy module stuff (functors)
- Polymorphic variants
- Mutable fields
- Catching exceptions; exceptions carrying values

Just don't need much of this for class
(nor do I use these features much)