**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 *)

```ocaml
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**

```ocaml
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!"
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**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ocamlc file.ml</td>
<td>compile to bytecodes (put in executable)</td>
</tr>
<tr>
<td>ocamlopt file.ml</td>
<td>compile to native (1-5x faster, no need in class)</td>
</tr>
<tr>
<td>ocamlc -i file.ml</td>
<td>print types of all top-level bindings (an interface)</td>
</tr>
<tr>
<td>ocaml</td>
<td>read-eval-print loop (see manual for directives)</td>
</tr>
<tr>
<td>ocamlprof, ocamldump, _</td>
<td>see the manual (probably unnecessary)</td>
</tr>
</tbody>
</table>

- Later: multiple files

---

**Installing, learning**

- Links from the web page:
  - [www.ocaml.org](http://www.ocaml.org)
  - The on-line manual (great reference)
  - An on-line book (less of a reference)
  - Installation/use instructions
- Post on Piazza with any install problems (soon!)
- Also ask questions
Types

• Every expression has one type. So far:

\[
\text{int} \quad \text{string} \quad \text{unit} \quad \text{t1}\rightarrow\text{t2} \quad '\text{a}'
\]

(* print_string : \text{string}\rightarrow\text{unit}, "\_" : \text{string} *)

let \(x\) = print_string "Hello, World!\n"

(* \(x\) : \text{unit} *)

...

(* ^ : \text{string} \rightarrow \text{string} \rightarrow \text{string} *)

let \(f\) \(x\) = print_string \((h \ ^ \ w)\)

(* \(f\) : \text{a} \rightarrow \text{unit} *)

let \(y1\) = \(f\) \(37\)

(let \(y1\) \(=\) \(f\) \(37\)

(* \(y1\) : \text{unit} *)

let \(y2\) = \(f\) \'(h \ ^ \ w)\)

(let \(y2\) \(=\) \(f\) \'(h \ ^ \ w)\)

(* \(y2\) : \text{unit} *)

let \(y3\) = \(y1\)

(let \(y3\) \(=\) \(y1\)

(* \(y3\) : \text{unit} *)

Explicit types

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

\[
\text{let } f \; x = \text{print_string} \; (h \ ^ \ w) \\
\text{let } g \; (x:\text{int}) = f \; x
\]

(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:

\[
\text{let rec}
\]

\[
\text{let rec } \text{forever} \; x = \text{forever} \; x
\]

(let \(\_\) \(=\) \(g\) \(37\)

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

Locals

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

let \(\text{quadruple}\) \(x\) =

  let \(\text{double}\) \(y\) = \(y\) + \(y\) in
  let \(\text{ans}\) = \(\text{double}\) \(x\) + \(\text{double}\) \(x\) in
  \(\text{ans}\)

let \(_\) =

  \text{print_string} ((\text{string}\_\text{of}\_\text{int}(\text{quadruple} \; 7)) \ ^ \"\n\"

Anonymous functions

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

let \(\text{quadruple2}\) \(x\) =

  (\text{fun} \; x \rightarrow x \ + \ x) \ + \ (\text{fun} \ x \rightarrow x \ + \ x) \ x

(let \(\text{quadruple3}\) \(x\) =

  let \(\text{double}\) = \(\text{fun}\) \(x\) \rightarrow \(x\) + \(x\) in
  \(\text{double}\) \(x\) + \(\text{double}\) \(x\)
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

OCaml tutorial, Boyana Norris

13

Multiple args, currying

let print_i_nl2 i f = print_i_nl (f i)

• Inferior style (fine, but Caml 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)

• Caml 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

```ocaml
closures are closed
return 11 is bound to a value

• 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 return_11 () = 11

```
guaranteed not to change the program

```

Another example

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

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

```ocaml
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 (pr1.first=5; pr1.second=6)
```

A type constructor for polymorphic data/code:

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

More polymorphic code

```ocaml
type 'a pair = {first : 'a; a_second : 'a}
let sum_pr f x = f x.first + f x.second
let pr2 = {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_n_l (sum_int_pair)
let _ = print_i_n_l (sum_int_pair)
let _ = print_i_n_l (sum_int_pair)
```

Mutation exists

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

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

Record types

```ocaml
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 (pr1.first=5; pr1.second=6)
```

A type constructor for polymorphic data/code:

```ocaml
type 'a pair = {first : 'a; a_second : 'a}
let sum_pr f x = f x.first + f x.second
let pr2 = {first = 3; a_second = 4} (*int pair*)
let _ = sum_pr pr2
  + sum_pr (fun x->x) (pr2.first=5; pr2.second=6)
```
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
- Caml does this directly; the tags are constructors
  - Type is called a datatype

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

Datatypes

| type food = Foo of int | Bar of int_pair |
| Bar of int * int | Quux |
| let foo3 = Foo (1 + 2) |
| let bar12 = Bar (pr) |
| let baz1_120 = Baz (1, fact 5) |
| let quux = Quux (* not much point in this *) |
| let is_a_foo x = match x with Foo i -> true |
| | Bar pr -> false |
| | Baz(i,j) -> false |
| | Quux -> false |

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 'a list = Null |
| Cons of 'a * 'a list |
| 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 list = Null |
| Cons of 'a * 'a list |
| let rec length list = (* 'a list -> int *) |
| match list with Null -> 0 |
| | Cons(x,rest) -> 1 + length rest |
Recursive functions

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

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

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]

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

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

```ocaml
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 _ 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 (x,y) = (match x with (x,y) -> x+y)
  - let f (x,y) = x + y
  - let f (x1,y1) (x2,y2) = x1 + y2
Fancy patterns example

```ocaml
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
  | _ -> raise 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
```

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

Module pragmatics

- `foo.mli` 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
  - Program-evaluation order
- See manual for `.cm[io]` files, `-c` flag, etc.

Module example

**foo.ml**

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

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

let even = int

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

**foo.mli**

```ocaml
(* choose to show *)
val get_int : t1 -> int

(* choose to hide *)
val makeEven : int -> even
val isEven : even -> bool
```

**bar.ml**

```ocaml
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.isEven(Foo.makeEven 17)
```

**foo.ml**

```ocaml
(* choose to show *)
val get_int : t1 -> int

(* choose to hide *)
val makeEven : int -> even
val isEven : even -> bool
```
Not the whole language

- Objects
- Loop forms
- 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)