An Introduction to OCaml

adapted from course by Stephen Edwards @ Columbia

The Basics

Functions

Tuples, Lists, and Pattern Matching

User-Defined Types

Modules and Compilation

A Complete Interpreter in Three Slides

Exceptions; Directed Graphs

Standard Library Modules

An Endorsement?

A PLT student accurately summed up using OCaml:

Never have I spent
so much time
writing so little
that does so much.

I think he was complaining, but I'm not sure.

Other students have said things like

It's hard to get it to compile, but once it compiles, it works.

Why OCaml?

- It's Great for Compilers
  I've written compilers in C++, Python, Java, and OCaml, and it's much easier in OCaml.
- It's Succinct
  Would you prefer to write 10 000 lines of code or 5 000?
- Its Type System Catches Many Bugs
  It catches missing cases, data structure misuse, certain off-by-one errors, etc. Automatic garbage collection and lack of null pointers makes it safer than Java.
- Lots of Libraries
  All sorts of data structures, I/O, OS interfaces, graphics, support for compilers, etc.
- Lots of Support
  Many websites, free online books and tutorials, code samples, etc.

OCaml in One Slide

Apply a function to each list element; save results in a list

"Is recursive"

Passing a function

Case splitting

Pattern Matching

Local name declaration

Polymorphic

List support

Types inferred

Recursion

Anonymous functions

```
# let rec map f = function
    [] -> []
  | head :: tail ->
      let r = f head in
      r ::  map f tail;;

val map :  ('a -> 'b) -> 'a list -> 'b list

# map (function x -> x + 3) [1;5;9];;

- :  int list = [4; 8; 12]
```

The Basics

Hello World in OCaml: Interpret or Compile

Create a "hello.ml" file:

```
print_endline "Hello World!"
```

Run it with the interpreter:

```
$ ocaml hello.ml
Hello World!
```

Compile a native executable and run:

```
$ ocamlopt -o hello hello.ml
$ ./hello
Hello World!
```

Use ocamlbuild (recommended):

```
$ ocamlbuild hello.native
$ ./hello.native
Hello World!
```

Hello World in OCaml: REPL

The interactive Read-Eval-Print Loop

```
$ ocaml
        OCaml version 4.02.3

# print_endline "Hello World!";;
Hello World!
- : unit = ()
# #use "hello.ml";;
Hello World!
- : unit = ()
# #quit;;
$
```

Double semicolons ;; mean "I'm done with this expression"

#quit terminates the REPL

Other directives enable tracing, modify printing, and display types and values

Use ledit ocaml or utop instead for better line editing (history, etc.)

Comments

OCaml

```
(* This is a multiline
   comment in OCaml*)

(* Comments
    (* like these *)
    do nest
*)

(* OCaml has no *)
(* single-line comments*)
```

C/C++/Java

```
/* This is a multiline
   comment in C  */

/* C comments
     /* do not
   nest
 */

// C++/Java also has
// single-line comments
```

Basic Types and Expressions

```
# 42 + 17;;
- : int = 59

# 42.0 +. 18.3;;
- : float = 60.3

# 42 + 60.3;;
Error: This expression has type
float but an expression was
expected of type int

# 42 + int_of_float 60.3;;
- : int = 102

# true || (3 > 4) && not false;;
- : bool = true

# "Hello " ^ "World!";;
- : string = "Hello World!"

# String.contains "Hello" 'o';;
- : bool = true

# ();;
- : unit = ()

# print_endline "Hello World!";;
Hello World!
- : unit = ()
```

Integers

Floating-point numbers

Floating-point operators must be explicit (e.g., +.)

Only explicit conversions, promotions (e.g., int_of_float)

Booleans

Strings

The unit type is like "void" in C and Java

Standard Operators and Functions

<table>
<tr><td><code>+ - * / mod</code></td><td>Integer arithmetic</td></tr>
<tr><td><code>+. -. *. /. **</code></td><td>Floating-point arithmetic</td></tr>
<tr><td><code>ceil floor sqrt exp</code>
<code>log log10 cos sin</code>
<code>tan acos asin atan</code></td><td>Floating-point functions</td></tr>
<tr><td><code>not && ||</code></td><td>Boolean operators</td></tr>
<tr><td><code>= <></code>
<code>== !=</code></td><td>Structual comparison (polymorphic)
Physical comparison (polymorphic)</td></tr>
<tr><td><code>< > <= >=</code></td><td>Comparisons (polymorphic)</td></tr>
</table>

Structural vs. Physical Equality

==, != Physical equality compares pointers

```
# 1 == 3;;
- : bool = false
# 1 == 1;;
- : bool = true
# 1.5 == 1.5;;
- : bool = false    (* Huh? *)
# let f = 1.5 in f == f;;
- : bool = true
# "a" == "a";;
- : bool = false    (* Huh? *)
# let a = "hello" in a == a;;
- : bool = true
```

=, <> Structural equality compares values

```
# 1 = 3;;
- : bool = false
# 1 = 1;;
- : bool = true
# 1.5 = 1.5;;
- : bool = true
# let f = 1.5 in f = f;;
- : bool = true
# "a" = "a";;
- : bool = true
```

Use structural equality to avoid headaches

If-then-else

$$\textbf{if}\ expr_1\ \textbf{then}\ expr_2\ \textbf{else}\ expr_3$$

If-then-else in OCaml is an expression. The *else* part is compulsory, $expr_1$ must be Boolean, and the types of $expr_2$ and $expr_3$ must match.

```
# if 3 = 4 then 42 else 17;;
- : int = 17

# if "a" = "a" then 42 else 17;;
- : int = 42

# if true then 42 else "17";;
This expression has type string but is here used with type int
```

Naming Expressions with *let*

let *name* = $expr_1$ **in** $expr_2$	Bind *name* to $expr_1$ in $expr_2$ only
let *name* = *expr*	Bind *name* to *expr* forever after

```
# let x = 38 in x + 4;;
  - : int = 42
# let x = (let y = 2 in y + y) * 10 in x;;
- : int = 40
# x + 4;;
Unbound value x
# let x = 38;;
val x : int = 38
# x + 4;;
- : int = 42
# let x = (let y = 2) * 10 in x;;
Error: Syntax error: operator expected.
# let x = 10 in let y = x;;
Error: Syntax error
```

Let is Not Assignment

Let can be used to bind a succession of values to a name. This is not assignment: the value disappears in the end.

```
# let a = 4 in
  let a = a + 2 in
  let a = a * 2 in
  a;;
- : int = 12

# a;;
Unbound value a
```

This looks like sequencing, but it is really data dependence.

Let is Really Not Assignment

OCaml picks up the values in effect where the function (or expression) is defined.

Global declarations are not like C's global variables.

```
# let a = 5;;
val a : int = 5

# let adda x = x + a;;
val adda : int -> int = <fun>

# let a = 10;;     (* a here is a diff var (copy) *)
val a : int = 10

# adda 0;;
- : int = 5        (* adda sees a = 5 *)

# let adda x = x + a;;
val adda : int -> int = <fun>

# adda 0;;
- : int = 10       (* adda sees a = 10 *)
```

Functions

Functions

A function is just another type whose value can be defined with an expression.

```
# fun x -> x * x;;
- : int -> int = <fun>
# (fun x -> x * x) 5;; (* function application  *)
- : int = 25

# fun x -> (fun y -> x * y);;
- : int -> int -> int = <fun>
# fun x y -> x * y;;    (* shorthand*  )
- : int -> int -> int = <fun>
# (fun x -> (fun y -> (x+1) * y)) 3 5;;
- : int = 20

# let square = fun x -> x * x;;
val square : int -> int = <fun>
# square 5;;
- : int = 25
# let square x = x * x;; (* shorthand*)
val square : int -> int = <fun>
# square 6;;
- : int = 36
```

Let is Like Function Application

let *name* = $expr_1$ **in** $expr_2$

(**fun** *name* **->** $expr_2$) $expr_1$

Both mean "$expr_2$, with *name* replaced by $expr_1$"

```
# let a = 3 in a + 2;;
- : int = 5
# (fun a -> a + 2) 3;;
- : int = 5
```

Semantically equivalent; `let` is easier to read

Recursive Functions

OCaml

```
let rec gcd a b =
  if a = b then
    a
  else if a > b then
    gcd (a - b) b
  else
    gcd a (b - a)
```

C/C++/Java

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

let rec allows for recursion

Use recursion instead of loops

Tail recursion runs efficiently in OCaml

Recursive Functions

By default, a name is not visible in its defining expression.

```
# let fac n = if n < 2 then 1 else n * fac (n-1);;
Unbound value fac
```

The *rec* keyword makes the name visible.

```
# let rec fac n = if n < 2 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>
# fac 5;;
- : int = 120
```

The *and* keyword allows for mutual recursion.

```
# let rec fac n = if n < 2 then 1 else n * fac1 n
  and fac1 n = fac (n - 1);;
val fac : int -> int = <fun>
val fac1 : int -> int = <fun>
# fac 5;;
- : int = 120
```

First-Class and Higher Order Functions

First-class functions: name them, pass them as arguments

```
# let appadd = fun f -> (f 42) + 17;;
val appadd : (int -> int) -> int = <fun>

# let plus5 x = x + 5;;
val plus5 : int -> int = <fun>

# appadd plus5;;
- : int = 64
```

Higher-order functions: functions that return functions

```
# let makeInc i = fun x -> x + i;;
val makeInc : int -> int -> int = <fun>

# let i5 = makeInc 5;;
val i5 : int -> int = <fun>

# i5 10;;
- : int = 15
```

Tuples, Lists, and Pattern Matching

Tuples

Pairs or tuples of different types separated by commas.

Very useful lightweight data type, e.g., for function arguments.

```
# (42, "Arthur");;
- : int * string = (42, "Arthur")
# (42, "Arthur", "Dent");;
- : int * string * string = (42, "Arthur", "Dent")

# let p = (42, "Arthur");;
val p : int * string = (42, "Arthur")
# fst p;;
- : int = 42
# snd p;;
- : string = "Arthur"

# let trip = ("Douglas", 42, "Adams");;
val trip : string * int * string = ("Douglas", 42, "Adams")
# let (fname, _, lname) = trip in (lname, fname);;
- : string * string = ("Adams", "Douglas")
```

Lists

```
(* Literals *)
[];;                (* The empty list *)
[1];;               (* A singleton list *)
[42; 16];;          (* A list of two integers *)

(* cons: Put something at the beginning *)
7 :: [5; 3];;       (* Gives [7; 5; 3] *)
[1; 2] :: [3; 4];; (* BAD: type error *)

(* concat: Append a list to the end of another *)
[1; 2] @ [3; 4];;   (* Gives [1; 2; 3; 4] *)

(* Extract first entry and remainder of a list *)
List.hd [42; 17; 28];; (* = 42 *)
List.tl [42; 17; 28];; (* = [17; 28] *)
```

The elements of a list must all be the same type.

`::` is very fast; `@` is slower—$O(n)$

Pattern: create a list with cons, then use *List.rev*.

Some Useful List Functions

Three great replacements for loops:

- `List.map f [a1; ... ;an] = [f a1; ... ;f an]`
  Apply a function to each element of a list to produce another list.
- `List.fold_left f a [b1; ...;bn] =`
  `f (...(f (f a b1) b2)...) bn`
  Apply a function to a partial result and an element of the list to produce the next partial result.
- `List.iter f [a1; ...;an] =`
  `begin f a1; ... ; f an; () end`
  Apply a function to each element of a list; produce a unit result.
- `List.rev [a1; ...; an] = [an; ... ;a1]`
  Reverse the order of the elements of a list.

List Functions Illustrated

```
# List.map (fun a -> a + 10) [42; 17; 128];;
- : int list = [52; 27; 138]

# List.map string_of_int [42; 17; 128];;
- : string list = ["42"; "17"; "128"]

# List.fold_left (fun s e -> s + e) 0 [42; 17; 128];;
- : int = 187

# List.iter print_int [42; 17; 128];;
4217128- : unit = ()

# List.iter (fun n -> print_int n; print_newline ())
   [42; 17; 128];;
42
17
128
- : unit = ()
# List.iter print_endline (List.map string_of_int [42; 17; 128]);;
42
17
128
- : unit = ()
```

Example: Enumerating List Elements

To transform a list and pass information between elements, use *List.fold_left* with a tuple:

```
# let (l, _) = List.fold_left
    (fun (l, n) e -> ((e, n)::l, n+1)) ([], 0) [42; 17; 128]
  in List.rev l;;
- : (int * int) list = [(42, 0); (17, 1); (128, 2)]
```

Result accumulated in the *(l, n)* tuple, *List.rev* reverses the result (built backwards) in the end. Can do the same with a recursive function, but *List.fold_left* separates list traversal from modification:

```
# let rec enum (l, n) = function
    []    -> List.rev l
  | e::tl -> enum ((e, n)::l, n+1) tl
  in
  enum ([], 0) [42; 17; 128];;
- : (int * int) list = [(42, 0); (17, 1); (128, 2)]
```

Pattern Matching

A powerful variety of multi-way branch that is adept at picking apart data structures. Unlike anything in C/C++/Java.

```
# let xor p = match p
  with (false, false) -> false
     | (false,  true) -> true
     | ( true, false) -> true
     | ( true,  true) -> false;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);;
- : bool = false
```

A name in a pattern matches anything and is bound when the pattern matches. Each may appear only once per pattern.

```
# let xor p = match p
  with (false, x) -> x
     | ( true, x) -> not x;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);;
- : bool = false
```

Case Coverage

The compiler warns you when you miss a case or when one is redundant (they are tested in order):

```
# let xor p = match p
  with (false, x) -> x
     | (x, true) -> not x;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(true, false)
val xor : bool * bool -> bool = <fun>

# let xor p = match p
  with (false, x) -> x
     | (true,  x) -> not x
     | (false, false) -> false;;
Warning U: this match case is unused.
val xor : bool * bool -> bool = <fun>
```

Wildcards

Underscore (_) is a wildcard that will match anything, useful as a default or when you just don't care.

```
# let xor p = match p
  with (true, false) | (false, true) -> true
     | _ -> false;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);;
- : bool = false
# xor (false, false);;
- : bool = false
# xor (true, false);;
- : bool = true

# let logand p = match p
  with (false, _) -> false
     | (true, x) -> x;;
val logand : bool * bool -> bool = <fun>
# logand (true, false);;
- : bool = false
# logand (true, true);;
- : bool = true
```

Pattern Matching with Lists

```
# let length = function (* let length = fun p -> match p with  *)
    [] -> "empty"
  | [_] -> "singleton"
  | [_; _] -> "pair"
  | [_; _; _] -> "triplet"
  | hd :: tl -> "many";;
val length : 'a list -> string = <fun>

# length [];;
- : string = "empty"

# length [1; 2];;
- : string = "pair"

# length ["foo"; "bar"; "baz"];;
- : string = "triplet"

# length [1; 2; 3; 4];;
- : string = "many"
```

Pattern Matching with *when* and *as*

The *when* keyword lets you add a guard expression:

```
# let tall = function
  | (h, s) when h > 180 -> s ^ " is tall"
  | (_, s) -> s ^ " is short";;
val tall : int * string -> string = <fun>
# List.map  tall [(183, "Stephen"); (150, "Nina")];;
- : string list = ["Stephen is tall"; "Nina is short"]
```

The *as* keyword lets you name parts of a matched structure:

```
# match ((3,9), 4) with
    (_ as xx, 4) -> xx
  | _ -> (0,0);;
- : int * int = (3, 9)
```

Application: Length of a list

```
let rec length l =
  if l = [] then 0 else 1 + length (List.tl l);;
```

Correct, but not very elegant. With pattern matching,

```
let rec length = function
   []    -> 0
 | _::tl -> 1 + length tl;;
```

Elegant, but inefficient because it is not tail-recursive (needs $O(n)$ stack space). Common trick: use an argument as an accumulator.

```
let length l =
  let rec helper len = function
      []    -> len
    | _::tl -> helper (len + 1) tl
  in helper 0 l
```

This is the code for the List.length standard library function.

OCaml Can Compile This Efficiently

OCaml source code

```
let length list =
  let rec helper len = function
      []    -> len
    | _::tl -> helper (len + 1) tl
  in helper 0 list
```

- Arguments in registers
- Pattern matching reduced to a conditional branch
- Tail recursion implemented with jumps
- LSB of an integer always 1

ocamlopt generates this x86 assembly

```
camlLength__helper:
.L101:
  cmpl  $1, %ebx        # empty?
  je    .L100
  movl  4(%ebx), %ebx # get tail
  addl  $2, %eax        # len++
  jmp   .L101
.L100:
  ret

camlLength__length:
  movl  %eax, %ebx
  movl  $camlLength__2, %eax
  movl  $1, %eax       # len = 0
  jmp   camlLength__helper
```