Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 11, Part 2 rev. 4: `ocaml`

CAML Resources

- The ocaml home site, containing distributions and documentation, may be found at:

  `http://caml.inria.fr/`

- A CAML user's manual in many formats is available at:
  `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`
  You probably want to keep a local copy in one of the available formats.

- Another reference:

  `http://www.ocaml-tutorial.org/` (Last change: 19 Jan 2007)

- Also see `http://caml.inria.fr/resources/`

'Outside' Interest

The following 'non-academic' entities have an interest in functional programming:

```
    * Microsoft has developed F# based on OCaml.

See: http://msdn.microsoft.com/en-us/fsharp

Available in Visual Studio 2010 and sure looks a lot like ocaml


    * Ericsson has developed Erlang (look it up)


    * Intel uses OCaml for verification

See: http://ocamlnews.blogspot.com/2009/12/formally-verifying-intels-floatin
```

CAML Pragmatics

(O)CAML is similar to that of LISP and ML, in the sense that:

- Interactive use (a compiler is available) is typical and used for incremental development; and
- A toplevel loop which performs type checking, argument evaluation (call by value), code compilation, evaluation and printing of the result is used.
- The language is based upon a *core language*, supplemented by a module system.
- Like ML, (O)CAML is dependent upon a *basis library*.

CAML Specifics and Distinctions

1. The command-line version of CAML is invoked by typing `ocaml` at the command prompt.
2. Input is case-sensitive.
3. Comments are the same format as in SML.
4. All variable names must begin with a lowercase letter; names beginning with a capital letter are reserved for constructors for user-defined data structures.
5. Recursive function definitions must include the `rec` designator.
6. Integer and Floating point operations are distinguished by separate symbols.
7. CAML allows multiple-argument functions to be represented in either curried or 'tuple' form, as the examples below indicate.

However the forms cannot be mixed.

8. CAML uses the semicolon (`;`) to delineate list elements.

9. The interactive system prompt is the `#` character and a pair of semicolons (;;) is the `ocaml` expression terminator. ;; is only necessary for interpreted mode use of `ocaml`. After an expression is entered, the system compiles it, executes it and prints the outcome of evaluation.

10. `ocaml` phrases are either simple expressions, or `let` definitions of identifiers which may be either values or functions.

11. (Like ML), explicit type declaration of function parameters is not necessary. `ocaml` will try to infer the type from usage in the function body.

12. Recursive functions *must* be defined with the `let rec` binding.

Simple CAML Examples

```
#1+2*3;;
- : int = 7

#let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265359

#let square x = x *. x;;
val square : float -> float = <fun>

#square(sin pi) +. square(cos pi);;
- : float = 1
```

CAML Interaction, Part 2

```
# let calc input = List.hd(input);;
val calc : 'a list -> 'a = <fun>
# calc([1;2;3]);;
- : int = 1
# let calct (input, input2) = List.hd(input)*List.hd(input2);;
val calct : int list * int list -> int = <fun>
```

Getting Out

There are several ways to exit the ocaml interpreter:

`CRTL-ALT-Z`

`#quit`

There exist other ways (can you find them?).

Recursion, Pattern Matching, List and `trace`

```
let rec member = function
     (x, []) -> false
   | (x, h::t) ->
       if (h = x)
         then true
         else member (x, t) ;;
```

With use:

```
# #use "member.caml";;
val member : 'a * 'a list -> bool = <fun>
# #trace member;;
member is now traced.
# member("a",["c";"b";"a"]);;
member <-- (<poly>, [<poly>; <poly>; <poly>])
member <-- (<poly>, [<poly>; <poly>])
member <-- (<poly>, [<poly>])
member --> true
member --> true
member --> true
- : bool = true
#
```

Built-In Functions

- Look at the Pervasives Module
- Some are in infix form:

```
# (+);;
- : int -> int -> int = <fun>
# 7+3;;
- : int = 10
```

- Can always ask for the signature:

```
# List.hd;;
- : 'a list -> 'a = <fun>
# sqrt;;
- : float -> float = <fun>
```

Defining (As Yet Unnamed) Functions

The 'match' form using the `function` keyword:

```
function  pattern_1  ->  expr_1
| ...
|  pattern_n  ->  expr_n
```

Notes:

- This expression evaluates to a functional value with **one argument**.
- From the manual (abbreviated/enhanced):

  When this function is applied to value v, this value is matched against each pattern $pattern_1$ to $pattern_n$.

  If one of these matchings succeeds (tested in order), then expression $expr_i$ associated with the matched pattern is evaluated, and this becomes the returned value of the function.

Additional Notes:

1. Look up the syntax of `expr`.
2. Many functions don't use the other match cases

The λ-calculus and CAML

```
# function p -> sqrt p;;
- : float -> float = <fun>

# ((function p -> sqrt p) 2.0);;
- : float = 1.4142135623730951
```

The Identitiy Function

```
# (function a -> a);;
- : 'a -> 'a = <fun>

# (function a -> a) [1;2];;
- : int list = [1; 2]

# (function a -> a) 3.14159;;
- : float = 3.14159

# (function a -> a) "hiMom";;
- : string = "hiMom"

# (function a -> a) (1,2,3);;
- : int * int * int = (1, 2, 3)
```

Be Careful of Type

```
# function p -> sqrt p;;
- : float -> float = <fun>

#  ((function p -> sqrt p) 2);;
This expression has type int but is here used with type float
```

Tuples

```
(* here's a tuple *)
# (1,2,3);;
- : int * int * int = (1, 2, 3)
```

Tuples as Function Arguments

```
# function p1 p2 p3 -> p1*p2*p3;; (* function must have 1 arg *)
Syntax error

(* solution--use a tuple as the argument *)

# function (p1,p2,p3) -> p1*p2*p3;;
- : int * int * int -> int = <fun>

#  ((function (p1,p2,p3) -> p1*p2*p3) (1,2,3));;
- : int = 6

(* you can mix tuples within tuples *)

# function (p1,p2,(p3,p4),p5) -> p1*p2+p3+p4/p5;;
- : int * int * (int * int) * int -> int = <fun>

#  (function (p1,p2,(p3,p4),p5) -> p1*p2+p3+p4/p5) (2,3,(1,4),2);;
- : int = 9
```

Definition Approach 2

This is Currying.

```
fun parameter_1 parameter_2 ... parameter_n ->  expr
```

Library Function Argument Interface

All the standard library functions are curried.

Example:

```
# List.nth;;
- : 'a list -> int -> 'a = <fun>

# List.nth [1;2;3;4] 0;;
- : int = 1

# List.nth ([1;2;3;4],0);;
This expression has type int list * int but is here
used with type 'a list
```

Example

```
(* alternative function defn (currying) *)

# fun p1 p2 p3 -> p1*p2*p3;;
- : int -> int -> int -> int = <fun>

# (fun p1 p2 p3 -> p1*p2*p3) 1 2 3;;
- : int = 6

 (* compare with previous function definition --
     notice signature difference! *)

# function (p1,p2,p3) -> p1*p2*p3;;
- : int * int * int -> int = <fun>
```

More Examples of Defining and Using λ-Functions

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

# ((function (x,y) -> x*y) (3,4));;
- : int = 12

# ((function x -> List.hd x) ["hi";"mom"]);;
- : string = "hi"

# ((function y -> List.tl y) ["hi";"mom"]);;
- : string list = ["mom"]
```

Naming the Functions (`let`)

- General `ocaml` syntax:

  `let <name-of-something> = <expr>`

- ```
  # let a = 10;;
  val a : int = 10
  ```

  Bad: imperative programming. Do not use.

- `let <fn-name> = <function-defn>`

- Example:

  ```
  # let a = function x -> x*x;;
  val a : int -> int = <fun>
  # a 7;;
  - : int = 49
  ```
  ```

The strategy becomes more complex since there are:

1. Aternative ways to define functions
2. The 'match' variant
3. The `rec` designator – required for recursive definitions
4. Shortcuts

Recursive Function Definitions

For a function to be defined recursively, you need to

1. Give it a name (so it can be referenced in the function body); and
2. Use the `rec` designator, i.e.,

   `let rec ...`

Example

```
let rec listMinrev2 = function x ->
if x==[] then
     failwith "listMinrev2 should not be used on an empty list"
              else
               if List.tl(x)==[] then List.hd(x)
               else min (List.hd x) (listMinrev2 (List.tl x));;
```

Imperative vs. Functional Example

```
(* imperative *)
# let w = 1;;
val w : int = 1
# let w = w + 1;;
val w : int = 2

(* functional programming example *)
# let wfun = function w -> w + 1;;
val wfun : int -> int = <fun>
# wfun w;;
- : int = 3
# w  ;;
- : int = 2
```

Examples: Defining Named Functions

What's wrong with the following function definition?

```
let rec recursiveFn2 = function (n) ->
if n==0 then []
        else
        sqrt n :: recursiveFn2 (n-1) ;;
```

```
(* here's the problem -- type inference:

#  #use "recursiveFn2.caml";;
File "recursiveFn2.caml", line 4, characters 13-14:
This expression has type int but is here used with type float
#
*)
```

Here's a solution:

```
let rec recursiveFn3 = function (n) ->
if n==0 then []
        else
        sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;


(* use:


# recursiveFn3 10;;
- : float list =
[3.16227766016837952; 3.; 2.82842712474619029; 2.64575131106459072;
 2.44948974278317788; 2.23606797749979; 2.; 1.73205080756887719;
 1.41421356237309515; 1.]
# recursiveFn3 6;;
- : float list =
[2.44948974278317788; 2.23606797749979; 2.; 1.73205080756887719;
 1.41421356237309515; 1.]
```

Tracing

```
#trace function-name;;
(* After executing this directive,
all calls to the function named function-name will be traced *

#untrace function-name;;
    Stop tracing the given function.

#untrace_all;;
    Stop tracing all functions traced so far.
```

Tracing the Recursion (#trace)

```
(* trace to show recursion (and possibly debug) *)

# #trace recursiveFn3;;
recursiveFn3 is now traced.
# recursiveFn3 6;;
recursiveFn3 <-- 6
recursiveFn3 <-- 5
recursiveFn3 <-- 4
recursiveFn3 <-- 3
recursiveFn3 <-- 2
recursiveFn3 <-- 1
recursiveFn3 <-- 0
recursiveFn3 --> []
recursiveFn3 --> [1.]
recursiveFn3 --> [1.4142135623730951; 1.]
recursiveFn3 --> [1.7320508075688772; 1.4142135623730951; 1.]
recursiveFn3 --> [2.; 1.7320508075688772; 1.4142135623730951; 1.]
```

```
recursiveFn3 -->
  [2.2360679774997898; 2.; 1.7320508075688772; 1.4142135623730951; 1.]
recursiveFn3 -->
  [2.4494897427831779; 2.2360679774997898; 2.; 1.7320508075688772;
   1.4142135623730951; 1.]
- : float list =
[2.4494897427831779; 2.2360679774997898; 2.; 1.7320508075688772;
 1.4142135623730951; 1.]
#
*)
```

Simplified Function Definition

```
# let rec recursiveFn3 = function (n) ->
if n==0 then []
        else
        sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;
val recursiveFn3 : int -> float list = <fun>

# recursiveFn3 4;;
- : float list = [2.; 1.7320508075688772; 1.4142135623730951; 1.]
```

Don't need the parens on the argument:

```
# let rec recursiveFn3 = function n ->
if n==0 then []
        else
        sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;
val recursiveFn3 : int -> float list = <fun>
# recursiveFn3 4;;
- : float list = [2.; 1.73205080756887719; 1.41421356237309515; 1.]
#
```

But further simplification possible:

```
let rec recursiveFn3 n =
if n==0 then []
        else
        sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;

# recursiveFn3 4;;
- : float list = [2.; 1.73205080756887719; 1.41421356237309515; 1.]
```

More of the Simplified Syntax

```
(* original *)

let rec boardform1D = function (n) ->
if n==0 then []
        else
        "empty" :: boardform1D (n-1) ;;

(* simpler *)

let rec boardform1Dr1 n =
if n==0 then []
        else
        "empty" :: boardform1Dr1 (n-1) ;;

(* original *)

let rec boardform2D = function (n,m) ->
if n==0 then []
```

```
        else
        boardform1D(m) :: boardform2D (n-1,m) ;;

(* simpler *)

let rec boardform2Dr1 (n,m) =
if n==0 then []
        else
        boardform1D(m) :: boardform2Dr1 (n-1,m) ;;
```

A Long (and Biased) List Recursion Example

```
# let rec recursiveFn = function (n) ->
if n==0 then []
        else
        "UVa" :: recursiveFn (n-1) ;;
val recursiveFn : int -> string list = <fun>
```

```
# recursiveFn 10;;
- : string list =
["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]

# recursiveFn (10);;
- : string list =
["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]

# (recursiveFn 10);;
- : string list =
["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]

# (recursiveFn (10));;
- : string list =
["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
#
```

Now trace it:

```
# #trace recursiveFn;;
recursiveFn is now traced.
# recursiveFn 10;;
recursiveFn <-- 10
recursiveFn <-- 9
recursiveFn <-- 8
recursiveFn <-- 7
recursiveFn <-- 6
recursiveFn <-- 5
recursiveFn <-- 4
recursiveFn <-- 3
recursiveFn <-- 2
recursiveFn <-- 1
recursiveFn <-- 0
recursiveFn --> []
recursiveFn --> ["UVa"]
recursiveFn --> ["UVa"; "UVa"]
recursiveFn --> ["UVa"; "UVa"; "UVa"]
recursiveFn --> ["UVa"; "UVa"; "UVa"; "UVa"]
```

```
recursiveFn --> ["UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
recursiveFn --> ["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
recursiveFn --> ["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
recursiveFn --> ["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
recursiveFn -->
  ["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
recursiveFn -->
  ["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
- : string list =
["UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"; "UVa"]
```

(Enhanced 'building a list' Example

```
let rec listBuild = function (size,element) ->
(* check for proper use *)
if size <= 0 then
     failwith "listBuild should only be used by trained personel"
else
  if size==1 then [element]
  else element:: listBuild(size-1,element);;
```

Don't Mix Curried vs. Tuple Function Designs!

```
# let calc input1 input2 = List.hd input1 * List.hd input2;;
val calc : int list -> int list -> int = <fun>

(* notice curried form above *)

# calc [1;2;3] [3;2;1];;
- : int = 3

# let calc2 (input1,input2) = List.hd input1 * List.hd input2;;
val calc2 : int list * int list -> int = <fun>
(* notice tuple form above *)

# calc2 ([1;2;3],[3;2;1]);;
- : int = 3

(* now some errors mixing curried and tuples *)

# calc ([1;2;3],[3;2;1]);;
```

```
This expression has type int list * int list but is here used with type
  int list

# calc2 [1;2;3] [3;2;1];;
This function is applied to too many arguments, maybe you forgot a ‘;’
```

match (or 'case')

Syntax:

```
match  expr
with  pattern_1  ->  expr_1
|        ...
|  pattern_n  ->  expr_n
```

Example

```
#  let isZero n = match n with
     0 -> true
   | _ -> false;;
val isZero : int -> bool = <fun>

# isZero 3;;
- : bool = false
# isZero 0;;
- : bool = true
```

’Linear Pattern Constraint’

CAML just matches.

```
let capture arg =
        match arg with
        (x,y,x,y) -> failwith "2 agents cannot be in same cell"
        |(x-1,y,x,y) -> 1
        |(x,y+1,x,y) -> 1
        |(x+1,y,x,y) -> 1
        |(x,y-1,x,y) -> 1
        |_ -> 0 ;;
```

From the manual: a variable cannot appear several times in a given pattern. In particular, there is no way to test for equality between two parts of a data structure using only a pattern (but `when` guards can be used for this purpose).

Let’s Add 2 Lists

```
let rec add2lists = fun x y ->
if (x=[] && y=[]) then []
(* in above you should check for equal length/end *)
   else ((List.hd x) + (List.hd y)) :: add2lists (List.tl x) (List.tl y);;

(* use:

# #use"add2lists.caml";;
val add2lists : int list -> int list -> int list = <fun>
# add2lists [1;2;3;4] [4;3;2;1];;
- : int list = [5; 5; 5; 5]
*)
```

ocaml Functions Must Be Defined Before Use

What's wrong witht he following file?

```
(* file 1.caml: fn 1 then 2 *)

let f1 = function (n) ->
  f2(n);;

let f2 = function(m) ->
m*m*m;;
```

```
$ ocaml
        Objective Caml version 3.12.1

# #use"file1.caml";;
File "file1.caml", line 4, characters 2-4:
Error: Unbound value f2
#
```

Let's try something different:

```
(* file 2.caml: fn 2 then 1 *)

let f2 = function(m) ->
m*m*m;;

let f1 = function (n) ->
  f2(n);;
```

```
# #use"file2.caml";;
val f2 : int -> int = <fun>
val f1 : int -> int = <fun>
# f1 6;;
- : int = 216
```

Mutually Recursive Functions

There is a potential problem with functions that recur through each other.

The problem is definition of one without a forward reference to the other.

Consider a file "odd-even.caml", containing the following example:

```
(*******************************************************
NOTE: THIS IS A PAIR OF MUTUALLY RECURSIVE FUNCTIONS AND
REQUIRES SPECIAL HANDLING IN ocaml.
*******************************************************)

let even n =
if (n==0) then true
          else odd (n-1);;

let odd m =
if (m==0) then false
          else even (n-1);;
```

(Attempted) use of this yields:

```
# #use "odd-even.caml";;
File "odd-even.caml", line 8, characters 15-18:
Unbound value odd
#
```

Why?

A solution[a] is to redefine the pair as mutually recursive functions.

See the ocaml manual Section 6.7.1.

```
let rec even n =
if (n==0) then true
          else odd (n-1)
and (* here's the mutual recursion *)
odd m =
if (m==0) then false
          else even (m-1);;
```

[a] Note that there are other ways to solve the general problem of 'odd or even'.

Use:

```
# #use"odd-even-mut-rec.caml";;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>

# even 6;;
- : bool = true
# odd 6;;
- : bool = false
# even 7
  ;;
- : bool = false
# odd 7;;
- : bool = true
```

Even more insight is gained by tracing the functions:

```
# #trace even;;
even is now traced.
# #trace odd;;
odd is now traced.

# even 6;;
even <-- 6
odd <-- 5
even <-- 4
odd <-- 3
even <-- 2
odd <-- 1
even <-- 0
even --> true
odd --> true
even --> true
odd --> true
even --> true
odd --> true
```

```
even --> true
- : bool = true

# odd 6;;
odd <-- 6
even <-- 5
odd <-- 4
even <-- 3
odd <-- 2
even <-- 1
odd <-- 0
odd --> false
even --> false
odd --> false
even --> false
odd --> false
even --> false
odd --> false
- : bool = false
# even 7;;
even <-- 7
```

```
odd <-- 6
even <-- 5
odd <-- 4
even <-- 3
odd <-- 2
even <-- 1
odd <-- 0
odd --> false
even --> false
odd --> false
even --> false
odd --> false
even --> false
odd --> false
even --> false
- : bool = false
# odd 7;;
odd <-- 7
even <-- 6
odd <-- 5
even <-- 4
```

```
odd <-- 3
even <-- 2
odd <-- 1
even <-- 0
even --> true
odd --> true
even --> true
odd --> true
even --> true
odd --> true
even --> true
odd --> true
- : bool = true
#
```

CAML I/O

- Many CAML functions for input and output are provided.
- Input and output are specified by input and output channels `in_channel` and `out_channel`, respectively. The defaults are stdin and stdout.
- Output functions are included in the Pervasives module and include `print_string` with signature `string -> unit`. This function prints an argument string on the standard output (a side effect) and returns type `unit`. Similarly, functions `print_int`, `print_float` and `print_newline` are also provided.

Some signatures:

```
# stdin;;
- : in_channel = <abstr>
# stdout;;
- : out_channel = <abstr>
# stderr;;
- : out_channel = <abstr>
# open_out;;
- : string -> out_channel = <fun>
# open_in;;
- : string -> in_channel = <fun>
# close_out;;
- : out_channel -> unit = <fun>
# close_in;;
- : in_channel -> unit = <fun>
# Printf.fprintf;;
- : out_channel -> ('a, out_channel, unit) format -> 'a = <fun>
# Scanf.fscanf;;
- : in_channel -> ('a, Scanf.Scanning.scanbuf, 'b) format -> 'a -> 'b = <fun
```

Using `Printf.printf`

- The `Printf` modules contains a number of more powerful printing functions intended for formatted printing, including function `printf`.
- A formatted string (called a `format`) is used.
- The structure of this string is similar to the arguments used in the `c` `printf` function. Highly recommended, esp. for `c` programmers).

Simplistic use:

```
(* see module Printf.printf *)
# Printf.printf "Hi Mom";;
Hi Mom- : unit = ()
```

More versatile use w/ format string:

```
# open Printf;;
# printf "\n\n\n%s\n\n%d" "And the answer is:" 10;;



And the answer is:

10- : unit = ()
#
```

```
(* here is a file writing example *)

# let my_chan = open_out "camlTest.out";;
val my_chan : out_channel = <abstr>

# Printf.fprintf my_chan "Hi Bob\n";;
- : unit = ()

(* still nothing in file-- until-- *)

# flush my_chan;;
- : unit = ()

(* now contents written to file
further extension----  *)

#  Printf.fprintf my_chan "\n %d   %d  %s\n" 10 20 "done";;
- : unit = ()
# flush my_chan;;
- : unit = ()
```

```
(* here are file contents so far: *)

Hi Bob

 10   20  done
```

More Printf.printf Examples

```
# Printf.printf "\n\n %i \n\n" 5;;


 5

- : unit = ()

#  Printf.printf "\n\n %f \n\n" 5.9;;


 5.900000

- : unit = ()
```

```
# open Printf;;  (* recall this is how we open a module *)

# printf "%f %f %f" 1.2 3.4 5.6;;
1.200000 3.400000 5.600000- : unit = ()
#

# printf "%s %d %f" "hi" 123 3.45;;
hi 123 3.450000- : unit = ()

# printf "%d %f %d %f" 1 2.3 4 5.6;;
1 2.300000 4 5.600000- : unit = ()

# printf "a=%d b=%f c=%f d=%d" 1 2.3 4.5 6;;
a=1 b=2.300000 c=4.500000 d=6- : unit = ()
```

```
# printf "\n first= %f\n second= %f\n third= %f\n" 1.2 3.4 5.6

 first= 1.200000
 second= 3.400000
 third= 5.600000
- : unit = ()

# printf "\n At iteration %d, the w^T= %f %f %f\n" 3 1.2 2.3 3

 At iteration 3, the w^T= 1.200000 2.300000 3.400000
- : unit = ()
```

Executing a Script

The `ocaml` command starts the toplevel system for Objective Caml.

This starts the interactive read-eval-print loop, but it does not need to be interactive, since it allows specification of a script file:

```
ocaml  [ script-file ]
```

Objects in CAML

Consider first the declaration of two CAML objects in the source file of Figure 1.

```
(* vehicle.caml *)

class vehicle =
object
val mutable name = "batmobile"
method print_name = name
end;;

class boat =
object
val mutable name = "leaky"
val mutable capacity = 4
method how_big = capacity
end;;
```

Figure 1: A Pair of Caml Objects

Results:

```
# #use "vehicle.caml";;
class vehicle :
  object val mutable name : string method print_name : string end
class boat :
  object
    val mutable capacity : int
    val mutable name : string
    method how_big : int
  end
# let my_vehicle = new vehicle;;
val my_vehicle : vehicle = <obj>
# my_vehicle#print_name;;
- : string = "batmobile"
# let my_boat = new boat;;
val my_boat : boat = <obj>
# my_boat#how_big;;
- : int = 4
# my_vehicle#how_big;;
This expression has type vehicle
It has no method how_big
# my_boat#print_name;;
This expression has type boat
It has no method print_name
#
```

Instead, consider the modification of the class definitions to create a hierarchy and employ inheritence. This is shown in Figure 2.

```
(* vehicle2.caml
   employs inheritence *)

class vehicle =
object
val mutable name = "batmobile"
method print_name = name
end;;

class boat =
object
inherit vehicle
val mutable capacity = 4
method how_big = capacity
end;;
```

Figure 2: Extension of the Class Structure of Figure 1 To Allow Inheritence

```
# #use "vehicle2.caml";;
class vehicle :
  object val mutable name : string method print_name : string end
class boat :
  object
    val mutable capacity : int
    val mutable name : string
    method how_big : int
    method print_name : string
  end
# let my_vehicle = new vehicle;;
val my_vehicle : vehicle = <obj>
# let my_boat = new boat;;
val my_boat : boat = <obj>
# my_vehicle#print_name;;
- : string = "batmobile"
# my_boat#print_name;;
- : string = "batmobile"
#
```

Figure 3: Behaviour of the OO System of Figure 2