

---

# MP 1 – Basic OCaml and Recursion

CS 421 – Fall 2006

Revision 1.0

**Assigned** August 29, 2006

**Due** September 5, 2006, 11:59 PM

**Extension** 48 hours (20% penalty)

---

## 1 Change Log

1.0 Initial Release.

## 2 Objectives and Background

The purpose of this MP is to test the student's ability to

- start up and interact with OCaml;
- define a function;
- write code that conforms to the type specified (this includes understand simple OCaml types, including functional ones);
- use pattern matching.

Another purpose of MPs is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to the MP problems. By the time of the exam, your goal is to be able to solve any of the following problems with pen and paper in less than 2 minutes.

## 3 Problems

**Note:** In the problems below, you do not have to begin your definitions in a manner identical to the sample code, which is present solely for guiding you better. However, you have to use the indicated name for your functions, and the functions will have to conform to any type information supplied, and to yield the same results as any sample executions given.

1. Write a function `times` which multiplies two real (float) numbers together.

```
# let times x y = ... ;;
val times : float -> float -> float = <fun>
# times 2.15 3.17 ;;
- : float = 6.8154999999999992
```

2. Write a function `unfold` that takes a list and returns a pair consisting of the first element in the list (i.e., the head) and the rest of the list (i.e., the tail). The case when the argument list is empty needs not be covered, so you may assume the argument list nonempty. (Note: in case you want to cover this NOT REQUIRED case too, be aware of the required polymorphic type of your function.)

```

# let unfold lst = ... ;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val unfold : 'a list -> 'a * 'a list = <fun>
# unfold [2 ; 3 ; 4] ;;
- : int * int list = (2, [3; 4])
# unfold [true ; false ; true] ;;
- : bool * bool list = (true, [false; true])

```

3. Write a function `nonempty_print` that takes a string as argument and prints it on the screen if it is nonempty, else prints "Empty String".

```

# let nonempty_print str = ... ;;
val nonempty_print : string -> unit = <fun>
# nonempty_print "Hello" ;;
Hello
- : unit = ()
# nonempty_print "" ;;
Empty String
- : unit = ()

```

4. Write a function `isnull` that returns true if a list is empty, and false otherwise, while at the same time printing "I am true" if its result is true and "I am false" if its result is false.

```

# let isnull lst = ... ;;
val isnull : 'a list -> bool = <fun>
# isnull [2 ; 5 ; 1] ;;
I am false
- : bool = false
# isnull [] ;;
I am true
- : bool = true

```

5. Write a function `multiply` that takes two pairs of real numbers  $(x, y)$  and  $(u, v)$  representing complex numbers and returns their complex product. (Namely, any pair  $(x, y)$  represents a complex number  $x + yi$ , and thus the product of  $(x, y)$  and  $(u, v)$  is  $(x * u - y * v, x * v + y * u)$ .)

```

# let multiply ...
val multiply : float * float -> float * float -> float * float = <fun>
# multiply (2., 3.) (1., 4.) ;;
- : float * float = (-10., 11.)

```

6. Write a function `apply` that takes as arguments a function  $f$  and two parameters  $x$  and  $y$  and applies  $f$  to the triple  $(x, y, x)$ .

```

# let apply f x y = ... ;;
(* we do not show you the type here; you should figure it out *)
# apply (fun (x, y, z) -> x * y + z) 2 5 ;;
- : int = 12
# apply (fun (x, lst, z) -> x :: z :: lst) true [] ;;
- : bool list = [true; true]

```

7. Write a function `partiallyApply` that takes a function  $f$  and an item  $x$  and returns the function  $y \mapsto f y x$ , i.e., the function which maps each  $y$  to  $f y x$ . In other words, `partiallyApply` takes a function  $f$  needing two arguments followed by its second argument  $x$  and partially applies  $f$  to  $x$ .

```

# let partiallyApply f x = ... ;;
val partiallyApply : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
# partiallyApply (+) 4 ;;
- : int -> int = <fun>
# partiallyApply (+) 4 3 ;;
- : int = 7

```

8. Write a function `app3` that takes a function needing three arguments followed by each of the three arguments, and applies the function to its arguments.

```

# let app3 f x y z = ... ;;
val app3 : ('a -> 'b -> 'c -> 'd) -> 'a -> 'b -> 'c -> 'd = <fun>
# app3 (fun x y z -> x < y && z) 3 4 true ;;
- : bool = true

```

### Extra credit problem

9. Write a function `permuteArgs` that takes a function  $f$  needing three arguments and a permutation  $perm$  of  $[1 ; 2 ; 3]$  (i.e., a list containing the elements 1, 2, 3 in an arbitrary order, e.g.,  $[3 ; 1 ; 2]$ ) and returns the function that maps  $x_1 x_2 x_3$  to  $f x_{p(1)} x_{p(2)} x_{p(3)}$ , where for each  $i \in \{1, 2, 3\}$ ,  $p(i)$  is the position of  $i$  in the permutation list  $perm$ . You may assume that the argument  $perm$  is indeed a permutation of  $[1 ; 2 ; 3]$ , and not anything else. You are encouraged to use nested pattern matching for this problem.

```

# let permuteArgs f perm = ... ;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[] (* several warnings of unused patterns may follow here *)
val permuteArgs : ('a -> 'a -> 'a -> 'b) -> int list -> 'a -> 'a -> 'a -> 'b =
  <fun>
# permuteArgs (fun x y z -> x + y * z) [2 ; 1 ; 3] 4 5 6 ;;
- : int = 29

```