1 Review of Pattern Matching

It turns out that basic OCaml pattern matching can happen without function, fun, and match. When you do a definition using let, the left-hand side of the assignment need not be a single variable; it can be a pattern. OCaml tries to evaluate the right-hand side; if it succeeds, and if the resulting value matches the pattern on the left-hand side, any identifiers in the pattern are assigned to the corresponding data objects in the value.

The following interactions with the OCaml REPL show some examples of valid syntax in OCaml, and some examples of invalid syntax in OCaml. If you’re ever wondering whether a given line of code will run in OCaml, we encourage you to try it out in the REPL, similarly to what we’ve done here!

```ocaml
# let x,y = 2,3;;
val x : int = 2
val y : int = 3
# let a,b,c = 10,20;;
Characters 12-17:
  ^^^^^
Error: This expression has type 'a * 'b
but an expression was expected of type 'c * 'd 7* 'e

# let [p] = 1::[];;
Characters 4-7:
  ^
```

## Contents

1. Review of Pattern Matching
2. Variant Types
3. `options` in OCaml
   3.1 Curry with Sugar
4. Sorting with Mergesort in OCaml
5. Summary

1 Review of Pattern Matching

It turns out that basic OCaml pattern matching can happen without function, fun, and match. When you do a definition using let, the left-hand side of the assignment need not be a single variable; it can be a pattern. OCaml tries to evaluate the right-hand side; if it succeeds, and if the resulting value matches the pattern on the left-hand side, any identifiers in the pattern are assigned to the corresponding data objects in the value.

The following interactions with the OCaml REPL show some examples of valid syntax in OCaml, and some examples of invalid syntax in OCaml. If you’re ever wondering whether a given line of code will run in OCaml, we encourage you to try it out in the REPL, similarly to what we’ve done here!

```ocaml
# let x,y = 2,3;;
val x : int = 2
val y : int = 3
# let a,b,c = 10,20;;
Characters 12-17:
  ^^^^^
Error: This expression has type 'a * 'b
but an expression was expected of type 'c * 'd 7* 'e

# let [p] = 1::[];;
Characters 4-7:
  ^
```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(_::_::_::_|[])
val p : int = 1
# let q::_ = ["hi"; "mom"; "dad"];;
Characters 4-8:
  let q::_ = ["hi"; "mom"; "dad"];;
  ~~~~

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[_]
val q : string = "hi"
# let [1; r; _ ] = [2; 4; 6];;
Characters 4-14:
  let [1; r; _ ] = [2; 4; 6];;
  ~~~~~~~~~~~

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(1::_::_::_::_|1::_::[]|1::[]|0::_|[])
Exception: Match_failure ("//toplevel//", 1, 4).

# type myvariant = A | B | C of int;;
type myvariant = A | B | C of int
# A;;
- : myvariant = A
# B 10;;
Characters 0-4:
  B 10;;
  ~~~~

Error: The constructor B expects 0 argument(s),
but is applied here to 1 argument(s)
# C 10;;
- : myvariant = C 10
# let [_; foo; _] = [C 10; C 20; A];;
Characters 4-15:
  let [_; foo; _] = [C 10; C 20; A];;
  ~~~~~~~~~~~

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(_::_::_::_::_|_::_::[]|_::[]|[])
val foo : myvariant = C 20
# let C myint = foo;;
Characters 4-11:
  let C myint = foo;;
  ~~~~
(A|B)
val myint : int = 20

2 Variant Types

When writing a procedure that can give you completely different kinds of answers—like “here’s the answer” versus “oops, failed”—the preferred solution is to use a variant type.

For example, if for some reason your procedure would sometimes give you a float and sometimes an int, you would use a variant type such as

```
type my_v_type = A of float | B of int
```

3 options in OCaml

If in particular your procedure sometimes gives you an answer (of some single type) and sometimes fails, you are supposed to use the following variant type, called an option, which is built into OCaml.

```
type 'a option = Some of 'a | None
```

This is useful for representing results when something can fail. For example, let’s define a careful float division that doesn’t accept division by zero.

```
let careful_div : int * int -> int option =
  function (a, b) ->
    match b with
    0 -> None
    | _ -> Some(a/b)
```

If you try to use `Some a/b` instead of `Some(a/b)`, it won’t work! The reason is because a constructor (like `Some`) is like a procedure—it binds tighter to its argument than any infix operator.

Consider a procedure that takes in a pair consisting of a data object and a list and finds what index in the list at which the data object is located.

```
let rec find_position : 'a * 'a list -> int =
  function (x, alod) ->
    match alod with
    [] -> ????????
    | hd::tl -> if hd = x then 1 else 1 + find_position (tl, item)
```

What should get returned when the search fails? You might think, well, in the case of success the procedure returns an int, so it had better return an int in the case of failure as well, because the return type has to be something. However, there is a more convenient way to write procedures which could fail such as this: by using options!

If we change our output of `find_position` to use options, our procedure looks like this:
let rec find_position : 'a * 'a list -> int option =  
  function (x, alod) ->  
  match alod with  
    [] -> None  
  | hd::tl -> if first = x  
      then Some(1)  
      else match find_position (x, rest) with  
        None -> None  
      | Some(y) -> Some(y + 1)

Note that using recursion with options is more complicated, as exemplified by the above coding example. It is not enough to add 1 to the recursive call of find_position as the output will not be of type int, it will be of type int option. To get around this, we must use pattern matching on the recursive call, as shown above.

3.1 Curry with Sugar

From last time:

let rec curried_containsP : 'a -> 'a list -> bool =  
  function x -> function alod ->  
  match alod with  
    [] -> false  
  | first::rest -> x = first || (curried_containsP x) rest

This can be written this way, using fun instead of function:

# let rec containsP : ('a -> 'a list -> bool) =  
  fun x mylist ->  
  match mylist with  
    [] -> false  
  | first::rest -> x = first || containsP x rest;;
val containsP : 'a -> 'a list -> bool = <fun>

The above procedure looks like it’s taking in two arguments. However, this is actually just syntactic sugar for currying! Note that you can’t use this with function, just with fun.

Just to let you know, you can do it even more concisely, without using fun or function at all:

let rec curried_containsP (x : 'a) (mylist : 'a list) -> bool =  
  match mylist with  
    [] -> false  
  | first::rest -> x=first || (curried_containsP x) rest
Similarly the map procedure can be written with currying using `fun` as follows:

```ocaml
let rec map : ('a -> 'b) -> 'a list -> 'b list =
  fun f mylist ->
  match mylist with
    [] -> []
  | first::rest -> f first :: map f rest
```

However, it can also be written even more concisely using the following syntactic sugar:

```ocaml
let rec map (f : 'a -> 'b) (mylist : 'a list) -> 'b list =
  match mylist with
    [] -> []
  | first::rest -> f first :: map f rest
```

### 4 Sorting with Mergesort in OCaml

You may remember our covering mergesort from Racket! The Racket code for mergesort is as follows:

```racket
(define merge
  (lambda (L1 L2)
    (cond
      ((empty? L1) L2)
      ((empty? L2) L1)
      ((< (car L1) (car L2)) (cons (car L1) (merge (cdr L1) L2)))
      (#true (cons (car L2) (merge L1 (cdr L2)))))))

(define merge-sort
  (lambda (L)
    (if (< (length L) 10)
      (insertion-sort L)
      (merge (merge-sort (take 10 L)) (merge-sort (drop 10 L))))))
```
Converting this code to OCaml (as well as re-defining `take` and `drop` in OCaml) we get the following:

```ocaml
let rec take : int * 'a list -> 'a list =  
  function (n, alod) -> 
    match n, alod with 
    0, _ -> [] 
  | n, hd::tl -> hd::take (n-1, tl) 

let rec drop : int * 'a list -> 'a list =  
  function (n, alod) -> 
    0, _ -> alod 
  | n, hd::tl -> drop (n-1, tl) 

let rec merge : 'a list * 'a list -> 'a list =  
  function alod1, alod2 -> 
    match alod1, alod2 with 
    [], _ -> alod2 
  | _, [] -> alod1 
  | hd1::tl1, hd2::tl2 -> 
    if hd1 < hd2 
      then hd1::(merge(tl1, alod2)) 
    else hd2::(merge(alod1, tl2)) 

let rec mergesort : 'a list -> 'a list =  
  function alod -> 
    match alod with 
    [] -> [] 
  | hd::[] -> hd::[] 
  | _ -> 
    merge (mergesort (take ((List.length alod)/2, alod)), 
           mergesort (drop ((List.length alod)/2, alod)))
```
There’s actually a way to save a little time in mergesort—use a procedure that splits a list into a first half and a second half.

```ocaml
let rec split : int * 'a list -> 'a list * 'a list = function k, alod ->
  match k, alod with
  0, _ -> [], alod
| _, hd::tl ->
    let firstlist, secondlist = split (k-1, tl) in
    hd::firstlist, secondlist

let rec mergesort2 : 'a list -> 'a list = function alod ->
  match alod with
  [] -> []
| hd::[] -> alod
| _ -> let firstlist, secondlist = split ((List.length alod)/2, alod) in
      merge (mergesort2 firstlist, mergesort2 secondlist)
```

5 Summary

- Pattern matching can happen without function, fun, and match.
- Options are a valueable data type which can be used when the result of a procedure cannot always be found.
- In order to save some time in mergesort, we can use a procedure which splits a list into two halves.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)