

# CMSC330: OCaml Data and Pattern Matching

Chris Kauffman

*Last Updated:  
Thu Sep 28 09:14:41 AM EDT 2023*

# Logistics

## Assignments

- ▶ Project 3 Due Fri 06-Oct: Regex → NFA → DFA
- ▶ **Quiz 2 on Fri 29-Sep in Discussion**  
REMINDER: Past Semester Quizzes available [under Resources](#) on class web page
- ▶ **Exam 1 on Thu 05-Oct**, covers topics through OCaml Pattern Matching

Reading: OCaml Docs <https://ocaml.org/docs>

- ▶ Tutorial: Your First Day with OCaml
- ▶ Tutorial: OCaml Language Overview

Goals: OCaml Overview

- ▶ Finish up Static Types / Type Inference
- ▶ Pattern Matching and Linked Lists

*Still owe you a Python Practicum video which will go out later*

# Announcements

None

# Overview and Plan

- ▶ OCaml has a variety of built-in data types like Linked Lists, Arrays, Tuples, Options, Refs, etc.
- ▶ Also makes it easy to create new types of data via **Records** (struct/object like) and **Algebraic Types** (something new)
  - ▶ Several provided types are actually combinations of Records and/or Algebraic Types with special syntax support
  - ▶ Ex: Lists/Options are Algebraic, Refs are Records, etc.
- ▶ **Pattern Matching** is often used with data types in OCaml to determine the structure of the data and make decisions on it
- ▶ OCaml allows for **destructuring** data in various ways that are slick

## Plan

- ▶ Pattern Matching basics with tuples
- ▶ Declaration of Records and pattern matching
- ▶ Built-in Linked Lists and pattern matching

# Pattern Matching in Programming Languages

- ▶ **Pattern Matching** as a programming language feature checks that data matches a certain structure the executes if so
- ▶ Can take many forms such as processing lines of input files that match a regular expression
- ▶ Pattern Matching in OCaml/ML combines
  - ▶ Case analysis: does the data match a certain structure
  - ▶ Destructure Binding: bind names to parts of the data
- ▶ Pattern Matching gives OCaml/ML a certain “cool” factor
- ▶ Associated with the `match/with` syntax as follows

```
match something with
| pattern1 -> result1      (* pattern1 gives result1 *)
| pattern2 ->              (* pattern 2... *)
    action;                (* does some side-effect action *)
    result2                 (* then gives result2 *)
| pattern3 -> result3      (* pattern3 gives result3 *)
```

## Simple Case Examples of match/with

In it's simplest form, match/with provides a nice multi-case conditional structure. Constant values can be matched.

`yoda_say bool` Conditionally execute code

`counsel mood` Bind a name conditionally

```
1 (* Demonstrate conditional action using match/with *)
2 let yoda_say bool =
3   match bool with
4   | true  -> printf "False, it is not.\n"
5   | false -> printf "Not true, it is.\n"
6 ;;
7
8 (* Demonstrate conditional binding using match/with *)
9 let counsel mood =
10   let message = (* bind message *)
11     match mood with (* based on mood's value *)
12     | "sad"    -> "Welcome to adult life"
13     | "angry"  -> "Blame your parents"
14     | "happy"  -> "Why are you here?"
15     | "ecstatic" -> "I'll have some of what you're smoking"
16     | s        -> "Tell me more about " ^ s (* match any string *)
17   in
18   print_endline message;
```

# Matching Tuples

- ▶ Tuples are declared via commas as in (a,b,c) or x,y
- ▶ Parens option but do improve readability
- ▶ Can be pattern matched in several ways as shown below

```
1 (* match_tuples.ml: examples of pattern matching with tuples *)
2 open Printf;;
3
4 let has_meaning pair =
5   match pair with
6   | (42,42) -> "full of meaning"
7   | (42,_)  -> "meaning first"      (* _ : don't care / ignore *)
8   | (_,42)  -> "meaning second"
9   | _       -> "there is no meaning"
10 ;;
11 let print_meaning a b c =
12   match a,b,c with
13   | 4,2,_   -> (* create tuple for pat-match *)
14               (* both patterns use same action *)
15   | _,4,2 -> printf "There is meaning\n";
16   | x,y,z -> printf "%d %d %d have no meaning\n" x y z;
17 ;;
18               (* x,y,z wild cards: match anything *)
```

Last case of (x,y,z) **destructures** the tuple to give its parts names which can be used in the action

## Exercise: Use match/with

Write the following functions using match/with in some way

```
val xor :  
  bool -> bool -> bool = <fun>  
# xor true false;;  
- : bool = true  
# xor true true;;  
- : bool = false  
  
(* return true if a/be are not  
   the same booleans *)  
let xor a b =  
  ...  
;;
```

```
val fib : int -> int = <fun>  
# fib 0;;  
- : int = 0  
# fib 2;;  
- : int = 1  
# fib 10;;  
- : int = 55  
  
(* recursive fibonacci via match *)  
let rec fib n =  
  ...  
;;
```



# Answers: Use match/with

Answers in match\_exercise.ml

```
val xor :  
  bool -> bool -> bool = <fun>  
# xor true false;;  
- : bool = true  
# xor true true;;  
- : bool = false  
  
(* return true if a/b/e are not  
   the same booleans *)  
let xor a b =  
  match a,b with  
  | true,false  
  | false,true -> true  
  | _ -> false  
;;
```

```
val fib : int -> int = <fun>  
# fib 0;;  
- : int = 0  
# fib 2;;  
- : int = 1  
# fib 10;;  
- : int = 55  
  
(* recursive fibonacci via match *)  
let rec fib n =  
  match n with  
  | 0 -> 0  
  | 1 -> 1  
  | n -> (fib (n-1)) + (fib (n-2))  
;;
```

# Terminology: Declarative Programming

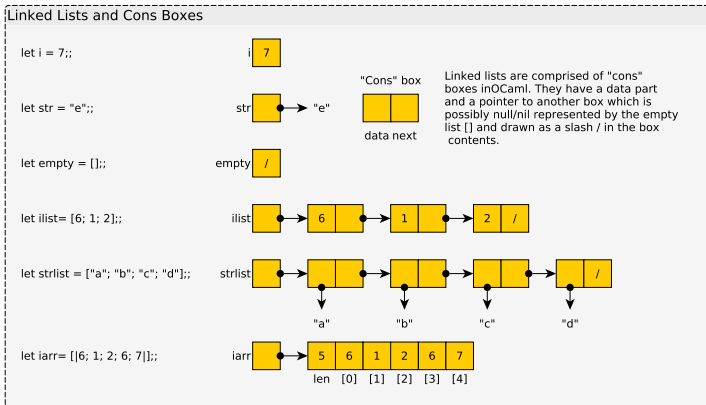
- ▶ **Declarative Programming** states how the output relates to the input, does not detail how to produce that output
- ▶ Example: Hypertext Markup Language (HTML) declares text, pictures, links should be on a web page but not exactly where, left to the **Browser Engine** to decide

```
<html> <body>
  
  <a href="https://clickthatbutton.com">
    Click that button
  </a>
  You know you want to.
</body> </html>
```

- ▶ Pattern matching has a Declarative feel to it: if data matches this pattern, do the following
- ▶ Exactly how the pattern is detected is left to OCaml's compiler; does guarantee **first-to-last checking** of patterns

# Lists in Functional Languages

- ▶ Long tradition of **Cons boxes** and **Singly Linked Lists** in Lisp/ML languages
- ▶ Immediate list construction of with square braces: [1;2;3]
- ▶ Note **unboxed** ints and **boxed** strings and lists in the below<sup>1</sup>



<sup>1</sup>“Boxed” means a pointer to data appears in the associated memory cell.

# List Parts with Head and Tail

- ▶ `List.hd list` : “head”, returns the first data element
- ▶ `List.tl list` : “tail”, returns the remaining list

## Accessing List Parts with List.hd and List.tl

```
let list1 = [6; 1; 2];;
```



```
let first = List.hd list1;;
```



```
let rest = List.tl list1;;
```



```
let restrest = List.tl rest;;
```



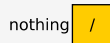
```
let last = List.hd restrest;;
```



```
let lenrr = List.length restrest;;
```



```
let nothing = List.tl restrest;;
```



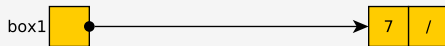
```
let nada = [];;
```



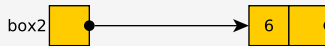
# List Construction with “Cons” operator ::

Constructing a list with successive "cons" applications

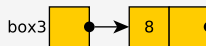
```
let box1 = 7 :: [];
```



```
let box2 = 6 :: box1;;
```



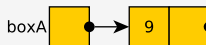
```
let box3 = 8 :: box2;;
```



```
let len = List.length box3;;
```



```
let boxA = 9 :: box2;;
```



```
let boxB = 4 :: box1;;
```



```
let lenA = List.length boxA;;
```



```
let lenB = List.length boxB;;
```



# Immutable Data

- ▶ Lists are **immutable** in OCaml
  - ▶ Cannot change list contents once created
  - ▶ `let` bindings are also immutable
- ▶ Immutable data is certainly a disadvantage if you want to change it (duh)
- ▶ Immutability creates some significant advantages
  - ▶ Easier reasoning: it won't change
  - ▶ Compiler may be able to optimize based on immutability
  - ▶ Can share structure safely to reduce memory usage
- ▶ Will have more to say later about trade-offs with immutability (sometimes called “persistent data”)

## Optional Exercise: List Construction/Decomposition

Fill in the Picture

```
let initial= [6; 1; 2];;
```



```
let listA = List.tl initial;;
```

listA

```
let listB = 7 :: listA;;
```

listB

```
let valX = List.hd listB;;
```

valX

```
let listC = (List.tl (List.tl listB));;
```

listC

```
let listD= 8 :: 5 :: 4 :: listC;;
```

listD

# Answers: List Construction/Decomposition

## Fill in the Picture: ANSWERS

let initial= [6; 1; 2];;

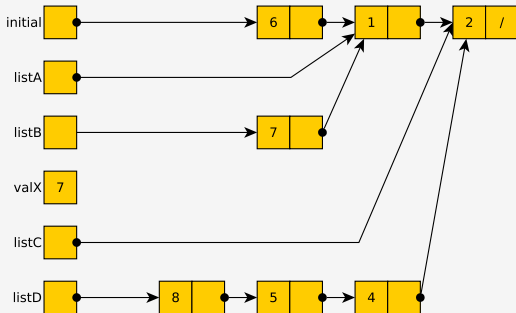
let listA = List.tl initial;;

let listB = 7 :: listA;;

let valX = List.hd listB;;

let listC = (List.tl (List.tl listB));;

let listD = 8 :: 5 :: 4 :: listC;;





# Patterns and Destructuring of Data

- ▶ Patterns can contain structure elements
- ▶ For lists, this is typically the Cons operator ::

```
1 let rec length_A list =  
2   match list with  
3   | []                -> 0  
4   | head :: tail -> 1 + (length_A tail)  
5 ;;
```

- ▶ Line 4 pattern binds names head/tail; compiler generates low level code like

```
let head = List.hd list in  
let tail = List.tl list in ...
```

- ▶ Pattern matching is relatively safe: the following will work and not generate any errors despite ordering of cases

```
1 let rec length_B list =  
2   match list with  
3   | head :: tail -> 1 + (length_B tail)  
4   | []          -> 0  
5 ;;
```

# Motivating Example: Summing Adjacent Elements

```
1 (* Create a list comprised of the sum of adjacent pairs of
2    elements in list. The last element in an odd-length list is
3    part of the return as is. *)
4 let rec sum_adj_ie list =
5     if list = [] then                                (* CASE of empty list *)
6         []                                           (* base case *)
7     else
8         let a = List.hd list in                    (* DESTRUCTURE list *)
9         let atail = List.tl list in                 (* bind names *)
10        if atail = [] then                          (* CASE of 1 elem left *)
11            [a]                                       (* base case *)
12        else                                         (* CASE of 2 or more elems left *)
13            let b = List.hd atail in                 (* destructure list *)
14            let tail = List.tl atail in              (* bind names *)
15            (a+b) :: (sum_adj_ie tail)              (* recursive case *)

# sum_adj_ie [1;2; 3;4; 5;6; 7];;
- : int list = [3; 7; 11; 7]

# sum_adj_ie [1;2; 3;4; 5;6; 7;8];;
- : int list = [3; 7; 11; 15]
```

- ▶ Paradigm: select **Cases** based on **Destructuring** list
- ▶ Note use of Cons `::` to build list recursively

# Pattern Matching on Lists Rocks

For structured data, pattern can improve case analysis markedly.

## if/else version of summing adjacent elements

```
1 let rec sum_adj_ie list =  
2   if list = [] then                                (* CASE of empty list *)  
3     []                                              (* base case *)  
4   else  
5     let a = List.hd list in                        (* DESTRUCTURE list *)  
6     let atail = List.tl list in                    (* bind names *)  
7     if atail = [] then                             (* CASE of 1 elem left *)  
8       [a]                                          (* base case *)  
9     else                                           (* CASE of 2 or more elems left *)  
10      let b = List.hd atail in                     (* destructure list *)  
11      let tail = List.tl atail in                  (* bind names *)  
12      (a+b) :: (sum_adj_ie tail)                  (* recursive case *)  
13 ;;
```

## match/with version of summing adjacent elements

```
1 let rec sum_adjacent list =  
2   match list with                                  (* case/destructure list separated by | *)  
3   | []      -> []                                  (* CASE of empty list *)  
4   | a :: [] -> [a]                                 (* CASE of 1 elem left *)  
5   | a :: b :: tail ->                             (* CASE of 2 or more elems left *)  
6     (a+b) :: sum_adjacent tail  
7 ;;
```

## Exercise: Swap Adjacent List Elements

Write the following function using **pattern matching**

```
let rec swap_adjacent list = ...;;  
(* Swap adjacent elements in a list. If the list is odd length,  
    the last element is dropped from the resulting list. *)
```

### REPL EXAMPLES

```
# swap_adjacent [1;2; 3;4; 5;6];;  
- : int list = [2; 1; 4; 3; 6; 5]  
# swap_adjacent ["a";"b"; "c";"d"; "e"];;  
- : string list = ["b"; "a"; "d"; "c"]  
# swap_adjacent [];;  
- : 'a list = []  
# swap_adjacent [5];;  
- : int list = []
```

For reference, solution to **summing** adjacent elements

```
1 let rec sum_adjacent list =  
2   match list with                (* case/destructure list separated by | *)  
3   | []                          -> []      (* CASE of empty list *)  
4   | a :: []                     -> [a]      (* CASE of 1 elem left *)  
5   | a :: b :: tail ->              (* CASE of 2 or more elems left *)  
6     (a+b) :: sum_adjacent tail  
7 ;;
```

## Answers: Swap Adjacent List Elements

```
1 (* Swap adjacent elements in a list. If the list is odd length,  
2    the last element is dropped from the resulting list. *)  
3 let rec swap_adjacent list =  
4   match list with  
5   | []                -> []                (* end of the line *)  
6   | a :: []           -> []                (* drop last elem *)  
7   | a :: b :: tail   ->                  (* two or more *)  
8       b :: a :: (swap_adjacent tail) (* swap order *)  
9 ;;
```

## Minor Details Associated with Pattern Matching

- ▶ First pattern: pipe | is optional
- ▶ Fall through cases: no action -> given, use next action
- ▶ Underscore \_ matches something, no name bound

### ▶ Examples of These

```
1 let cheap_counsel mood =
2   match mood with
3     "empty" ->                                (* first pipe | optional *)
4     printf "Eat something.\n";
5   | "happy" | "sad" | "angry" ->              (* multiple cases, same action *)
6     printf "Tomorrow you won't feel '%s'\n" mood;
7   | _ ->                                       (* match anything, no binding *)
8     printf "I can't help with that.\n";
9 ;;
```

- ▶ Arrays work in pattern matching but there is no size generalization as there is with list head/tail : arrays aren't defined inductively thus don't usually process them with pattern matching (see code in match\_basics.ml)

# Compiler Checks

Compiler will check patterns and warn if the following are found

- ▶ **Duplicate cases:**

only one can be used so the other is unreachable code

- ▶ **Missing cases:** data may not match any pattern and an exception will result

```
> cat -n match_problems.ml
1  (* duplicate case "hi": 2nd case not used *)
2  let opposites str =
3      match str with
4      | "hi" -> "bye"
5      | "hola" -> "adios"
6      | "hi" -> "oh god, it's you"
7      | s -> s^" is it's own opposite"
8      ;;
9
10 (* non-exhaustive matching *)
11 let list_size list =
12     match list with
13     | [] -> "0"
14     | a :: b :: [] -> "2"
15     | a :: b :: c :: [] -> "3"
16     ;; (* missing longer lists *)
> ocamlc -c match_problems.ml
File "match_problems.ml", line 6
Warning 11: this match case is unused.

File "match_problems.ml", line 12
Warning 8: this pattern-matching is not
exhaustive. Here is an example of a
case that is not matched: (_::_::_::_::_|_::[])
```

# Limits in Pattern Matching

- ▶ Patterns have limits
  - ▶ Can bind names to structural parts
  - ▶ Check for constants like [], 1, true, hi
  - ▶ Names in patterns are **always new bindings**
  - ▶ Cannot compare pattern bound name to another binding
  - ▶ Can't call functions in a pattern
- ▶ Necessitates use of conditionals in a pattern to further distinguish cases

```
1 (* Count how many times elem appears in list *)
2 let rec count_occur elem list =
3   match list with
4   | [] -> 0
5   | head :: tail ->      (* pattern doesn't compare head and elem *)
6       if head=elem then  (* need an if/else to distinguish *)
7         1 + (count_occur elem tail)
8       else
9         count_occur elem tail
10  ;;
```

- ▶ If only there were a nicer way... and there is.



## when Guards in Pattern Matching

- ▶ A pattern can have a when clause, like an if that is evaluated as part of the pattern
- ▶ Useful for checking additional conditions aside from structure

```
1 (* version that uses when guards *)
2 let rec count_occur elem list =
3   match list with
4   | [] -> 0
5   | head :: tail when head=elem -> (* check equality in guard *)
6     1 + (count_occur elem tail)
7   | head :: tail -> (* not equal, alternative *)
8     count_occur elem tail
9 ;;
10 (* Return strings in list longer than given
11    minlen. Calls functions in when guard *)
12 let rec strings_longer_than minlen list =
13   match list with
14   | [] -> []
15   | str :: tail when String.length str > minlen ->
16     str :: (strings_longer_than minlen tail)
17   | _ :: tail ->
18     strings_longer_than minlen tail
19 ;;
```

- ▶ Pattern Matching and Guards make for powerful programming

## Exercise: Convert to Patterns/Guards

Convert the following function (helper) to make use of match/with and when guards.

```
1 (* Create a list of the elements between the indices start/stop in the
2    given list. Uses a nested helper function for most of the work. *)
3 let elems_between start stop list =
4   let rec helper i lst =
5     if i > stop then
6       []
7     else if i < start then
8       helper (i+1) (List.tl lst)
9     else
10      let first = List.hd lst in
11      let rest = List.tl lst in
12      let sublist = helper (i+1) rest in
13      first :: sublist
14   in
15   helper 0 list
16 ;;
```

## Answers: Convert to Patterns/Guards

- ▶ Note the final “catch-all” pattern which causes failure
- ▶ Without it, compiler reports the pattern `[]` may not be matched

```
1 (* version of elems_between which uses match/with and when guards. *)
2 let elems_between start stop list =
3   let rec helper i lst =
4     match lst with
5     | _           when i > stop  -> []
6     | _ :: tail   when i < start -> helper (i+1) tail
7     | head :: tail                -> head :: (helper (i+1) tail)
8     | _                -> failwith "out of bounds"
9   in
10  helper 0 list
11 ;;
```

# Pattern Match Wrap

- ▶ Will see more of pattern matching as we go forward
- ▶ Most things in OCaml can be pattern matched, particularly symbolic data types for structures

```
1 open Printf;;
2
3 (* match a pair and swap elements *)
4 let swap_pair (a,b) =
5   let newpair = (b,a) in
6   newpair
7 ;;
8
9 (* 3 value kinds possible *)
10 type fruit = Apple | Orange | Grapes of int;;
11
12 (* match a fruit *)
13 let fruit_string f =
14   match f with
15   | Apple -> "you have an apple"
16   | Orange -> "it's an orange"
17   | Grapes(n) -> sprintf "%d grapes" n
18   ;;
```