CMSC330: OCaml Data and Pattern Matching

Chris Kauffman

Last Updated:
Tue Oct 3 09:16:24 AM EDT 2023

Logistics

Assignments

- Project 3 Due Fri 06-Oct: Regex $\to$ NFA $\to$ DFA
- **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 today

Announcements

All Online Lecture Quizzes Due Mon 11:59pm

- If I misspoke at some point, apologies for the confusion it created
- Lecture quizzes will **always** be released by Friday mornings
- Always be due following Mon 11:59pm
  Unless we announce to the entire class otherwise...

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 **Variant / 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
- Built-in Linked Lists and pattern matching
- Post-Exam: Records and (Variant) Data Types

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

```
(* Demonstrate conditional action using match/with *)
let yoda_say bool =
  match bool with
  | true  -> printf "False, it is not.\n"
  | false -> printf "Not true, it is.\n"
;;

(* Demonstrate conditional binding using match/with *)
let counsel mood =
  let message =                                    (* bind message *)
    match mood with                                (* based on mood's value *)
    | "sad"      -> "Welcome to adult life"
    | "angry"    -> "Blame your parents"
    | "happy"    -> "Why are you here?"
    | "ecstatic" -> "I'll have some of what you're smoking"
    | s          -> "Tell me more about "^s   (* match any string *)
  in
  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

```
(* match_tuples.ml: examples of pattern matching with tuples *)
open Printf;;

let has_meaning pair =
  match pair with
  | (42,42) -> "full of meaning"
  | (42,_)  -> "meaning first"        (* _ : don't care / ignore *)
  | (_,42)  -> "meaning second"
  | _       -> "there is no meaning"
;;
let print_meaning a b c =
  match a,b,c with                    (* create tuple for pat-match *)
  | 4,2,_                             (* both patterns use same action *)
  | _,4,2 -> printf "There is meaning\n";
  | x,y,z -> printf "%d %d %d have no meaning\n" x y z;
;;               (* 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/b 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/be 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

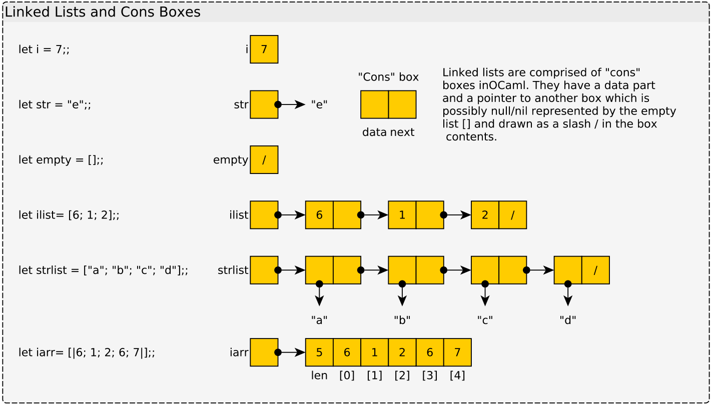
- **Declarative Programming** states how the output relates to the input, does not detail how to produce that output
- Example: Hypertext Markup Language (HTML) declares text, pictures, links should be on a web page but not exactly where, left to the Browser Engine to decide

```
<html> <body>
    <img src="button.jpg"/>
    <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[1]



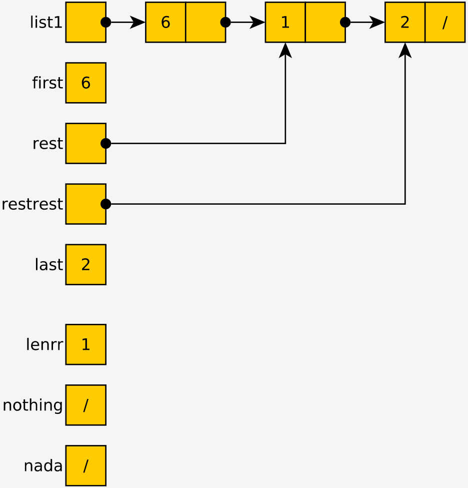
[1] "Boxed" means a pointer to data appears in the associated memory cell.

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

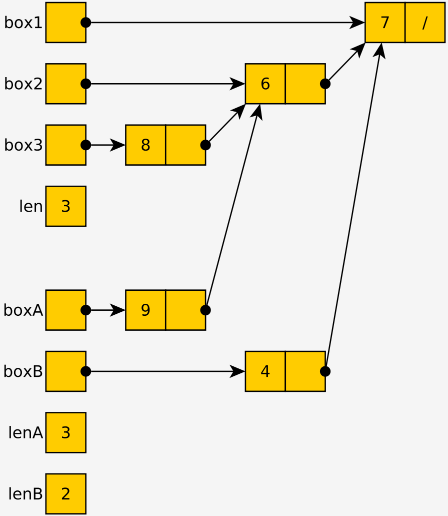
```
let list1 = [6; 1; 2];;

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;;
```



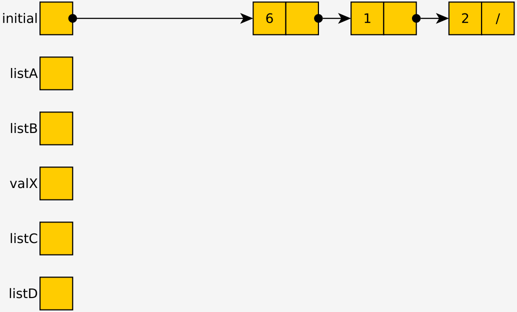
Immutable Data

- 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

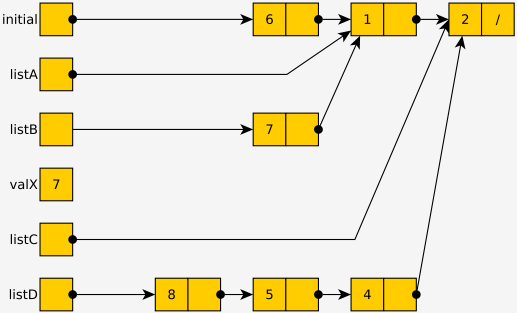
```
let initial= [6; 1; 2];;

let listA = List.tl initial;;

let listB = 7 :: listA;;

let valX = List.hd listB;;

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

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



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

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

Motivating Example: Summing Adjacent Elements

```
(* Create a list comprised of the sum of adjacent pairs of
   elements in list. The last element in an odd-length list is
   part of the return as is. *)
let rec sum_adj_ie list =
  if list = [] then                     (* CASE of empty list *)
    []                                  (* base case *)
  else
    let a = List.hd list in             (* DESTRUCTURE list *)
    let atail = List.tl list in         (* bind names *)
    if atail = [] then                  (* CASE of 1 elem left *)
      [a]                               (* base case *)
    else                                (* CASE of 2 or more elems left *)
      let b = List.hd atail in          (* destructure list *)
      let tail = List.tl atail in       (* bind names *)
      (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

```
let rec sum_adj_ie list =
  if list = [] then                    (* CASE of empty list *)
    []                                 (* base case *)
  else
    let a = List.hd list in            (* DESTRUCTURE list *)
    let atail = List.tl list in        (* bind names *)
    if atail = [] then                 (* CASE of 1 elem left *)
      [a]                              (* base case *)
    else                               (* CASE of 2 or more elems left *)
      let b = List.hd atail in         (* destructure list *)
      let tail = List.tl atail in      (* bind names *)
      (a+b) :: (sum_adj_ie tail)       (* recursive case *)
;;
```

match/with version of summing adjacent elements

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

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

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

Answers: Swap Adjacent List Elements

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

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

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

- 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

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

- 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

```
(* version that uses when guards *)
let rec count_occur elem list =
  match list with
  | [] -> 0
  | head :: tail when head=elem -> (* check equality in guard *)
     1 + (count_occur elem tail)
  | head :: tail ->                (* not equal, alternative *)
     count_occur elem tail
;;
(* Return strings in list longer than given
   minlen. Calls functions in when guard *)
let rec strings_longer_than minlen list =
  match list with
  | [] -> []
  | str :: tail when String.length str > minlen ->
     str :: (strings_longer_than minlen tail)
  | _ :: tail ->
     strings_longer_than minlen tail
;;
```

- Pattern Matching and Guards make for powerful programming

(Optional) Exercise: Convert to Patterns/Guards

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

```
(* Create a list of the elements between the indices start/stop in the
   given list. Uses a nested helper function for most of the work. *)
let elems_between start stop list =
  let rec helper i lst =
    if i > stop then
      []
    else if i < start then
      helper (i+1) (List.tl lst)
    else
      let first = List.hd lst in
      let rest =  List.tl lst in
      let sublst = helper (i+1) rest in
      first :: sublst
  in
  helper 0 list
;;
```

Answers: Convert to Patterns/Guards

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

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

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

```
open Printf;;

(* match a pair and swap elements *)
let swap_pair (a,b) =
  let newpair = (b,a) in
  newpair
;;

(* 3 value kinds possible *)
type fruit = Apple | Orange | Grapes of int;;

(* match a fruit  *)
let fruit_string f =
  match f with
  | Apple -> "you have an apple"
  | Orange -> "it's an orange"
  | Grapes(n) -> sprintf "%d grapes" n
;;
```