CSCI 2041: Pattern Matching Basics

Chris Kauffman

Last Updated:
Fri Sep 28 08:52:58 CDT 2018

Logistics

Reading

- OCaml System Manual: Ch 1.4 - 1.5
- Practical OCaml: Ch 4

Goals

- Code patterns
- Pattern Matching

Assignment 2

- Demo in lecture
- Post today/tomorrow

Next Week

- Mon: Review
- Wed: Exam 1
- Fri: Lecture

Consider: Summing Adjacent Elements

```
(* match_basics.ml: basic demo of pattern matching *)

(* 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 *)
```

The above function follows a common paradigm:

- Select between **Cases** during a computation
- Cases are based on structure of data
- Data is **Destructured** to bind names to parts of it

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;
```

Patterns and Destructuring

- 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
;;
```

Compare: if/else versus match/with version

Pattern matching often reduces improves clarity by reducing length

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

- 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": second case not use
  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: missing larger
 11  let list_size list =
 12    match list with
 13    | [] -> "0"
 14    | a :: b :: [] -> "2"
 15    | a :: b :: c :: [] -> "3"
 16  ;;

> 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 patter
- 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

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
;;
```