Pattern Matching

CS3100 Fall 2019

Review

Previously:

- Tuples, Records, Variants
- Polymorphism
- Lists, Option

This lecture:

- Pattern Matching

Pattern Matching

- Pattern matching is data deconstruction
  - Match on the *shape* of data
  - Extract part(s) of data

Syntax

```
match e with
| p1 -> e1
| p2 -> e2
...
| pn -> en
```

- p1 ... pn are patterns.

Pattern Matching on Lists

```
type 'a list = [] | :: of 'a * 'a list
```

- For lists, the patterns allowed follow from the constructors
  - The pattern `[]` matches the value `[]`.

- The patterh `h::t`
  - matches `2::[]`, binding `h` to `2` and `t` to `[]`.
  - matches `2::3::[]`, binding `h` to `2` and `t` to `3::[]`.
- The pattern `_` is a **wildcard pattern** and matches anything.

In [1]:

```
let list_status l =
  match l with
  | [] -> print_endline "The list is empty"
  | h::t -> Printf.printf "The list is non-empty. Head = %d\n%!" h
```

Out[1]:

```
val list_status : int list -> unit = <fun>
```

In [2]:

```
list_status []
```

```
The list is empty
```

Out[2]:

```
- : unit = ()
```

In [3]:

```
list_status [1;2;3]
```

```
The list is non-empty. Head = 1
```

Out[3]:

```
- : unit = ()
```

In [4]:

```
list_status (2::[3;4])
```

```
The list is non-empty. Head = 2
```

Out[4]:

```
- : unit = ()
```

Why pattern matching is THE GREATEST

1. You cannot forget to match a case (Exhaustivity warning)

In [5]:

```
let list_status l =
  match l with
  | [] -> print_endline "The list is empty"
  | h1::h2::t -> Printf.printf "The list is non-empty. 2nd element = %d\
```

```
File "[5]", line 2, characters 2-139:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
_::[]
```

Out[5]:

```
val list_status : int list -> unit = <fun>
```

Why pattern matching is THE GREATEST

1. You cannot forget to match a case (Exhaustivity warning)
2. You cannot duplicate a case (Unused case warning)

In [6]:

```
let list_status l =
  match l with
  | [] -> print_endline "The list is empty"
  | h::t -> Printf.printf "The list is non-empty. Head = %d\n%!" h
  | h1::h2::t -> Printf.printf "The list is non-empty. 2nd element = %d\
```

```
File "[6]", line 5, characters 4-13:
Warning 11: this match case is unused.
```

Out[6]:

```
val list_status : int list -> unit = <fun>
```

Why pattern matching is THE GREATEST

1. You cannot forget to match a case (Exhaustivity warning)
2. You cannot duplicate a case (Unused case warning)

Pattern matching leads to elegant, concise, beautiful code

Length of list

In [7]:

```
let rec length l =
  match l with
  | [] -> 0
  | h::t -> 1 + length t
```

Out[7]:

```
val length : 'a list -> int = <fun>
```

What is wrong with this code?

Length of list (tail recursive)

In [8]:

```
let rec length' l acc =
  match l with
  | [] -> acc
  | h::t -> length' t (1+acc)

let length l = length' l 0
```

Out[8]:

```
val length' : 'a list -> int -> int = <fun>
```

Out[8]:

```
val length : 'a list -> int = <fun>
```

In [9]:

```
length [1;2;3;4]
```

Out[9]:

```
- : int = 4
```

Match ordering

The patterns are matched in the order that they are written down.

In [10]:

```
let is_empty l =
  match l with
  | [] -> true
  | _ -> false
```

Out[10]:

```
val is_empty : 'a list -> bool = <fun>
```

Exercise

Implement the reverse of a list.

In [11]:

```
let rev_list l = failwith "not implemented"
```

Out[11]:

```
val rev_list : 'a -> 'b = <fun>
```

In [12]:

```
assert (rev_list [1;2;3] = [3;2;1])
```

```
Exception: Failure "not implemented".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "[12]", line 1, characters 8-24
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Exercise

Implement the append of two lists.

In [13]:

```
[1;2;3] @ [4;5;6]
```

Out[13]:

```
- : int list = [1; 2; 3; 4; 5; 6]
```

In [14]:

```
let append l1 l2 = failwith "not implemented"
```

Out[14]:

```
val append : 'a -> 'b -> 'c = <fun>
```

In [15]:

```
assert (append [1;2;3] [4;5;6] = [1;2;3;4;5;6])
```

```
Exception: Failure "not implemented".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "[15]", line 1, characters 8-30
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Nested Matching

In [16]:

```
type color = Red | Green | Blue

type point = {x : int; y : int}

type shape =
  | Circle of point * float (* center, radius *)
  | Rect of point * point   (* lower-left, upper-right *)
  | ColorPoint of point * color
```

Out[16]:

```
type color = Red | Green | Blue
```

Out[16]:

```
type point = { x : int; y : int; }
```

Out[16]:

```
type shape =
    Circle of point * float
  | Rect of point * point
  | ColorPoint of point * color
```

Nested Matching

Is the first shape in a list of shapes a red point?

In [17]:

```
let is_hd_red_circle l =
  match l with
  | ColorPoint(_,Red)::_ -> true
  | _ -> false
```

Out[17]:

```
val is_hd_red_circle : shape list -> bool = <fun>
```

Nested Matching

Print the coordinates if the point is green.

In [18]:

```
let rec print_green_point l =
  match l with
  | [] -> ()
  | ColorPoint({x;y}, Green)::tl ->
      Printf.printf "x = %d y = %d\n%!" x y;
      print_green_point tl
  | _::tl -> print_green_point tl
```

Out[18]:

```
val print_green_point : shape list -> unit = <fun>
```

In [19]:

```
print_green_point [Rect ({x=1;y=1},{x=2;y=2});
                   ColorPoint ({x=0;y=0}, Green);
                   Circle ({x=1;y=3}, 5.4);
                   ColorPoint ({x=4;y=6}, Green)]
```

```
x = 0 y = 0
x = 4 y = 6
```

Out[19]:

```
- : unit = ()
```

When do you use ";"

When you evaluate an expression just the effect, you can sequence the expression with a semi-colon.

```
let () = print_endline "Hello, world!" in
e
```

is equivalent to:

```
print_endline "Hello, world!";
e
```

Latter is considered better style.

Exceptions

- OCaml has support for exceptions.
  - Similar to the ones found in C++ & Java.
- Exceptions are (mostly) just variants.

```
type exn
exception MyException of string
```

- The type `exn` is an **extensible variant**.
  - New constructors of this type can be added after its original declaration.
- Exceptions are raised with `raise e` where `e` is of type `exn`.
- Handling exceptions is similar to pattern matching.

Find the green point

Given a list of shapes return a point whose colour is green. Otherwise, raise `NoGreenPoint` exception.

In [20]:

```
exception NoGreenPoint

let rec find_green_point l =
  match l with
  | [] -> raise NoGreenPoint
  | h::tl ->
    match h with
    | ColorPoint (_, Green) -> h
    | _ -> find_green_point tl
```

Out[20]:

```
exception NoGreenPoint
```

Out[20]:

```
val find_green_point : shape list -> shape = <fun>
```

Find the green point

In [21]:

```
find_green_point []
```

```
Exception: NoGreenPoint.
Raised at file "[20]", line 5, characters 16-28
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

In [22]:

```
find_green_point [Rect ({x=1;y=1},{x=2;y=2}); ColorPoint ({x=0;y=0}, Gre
```

Out[22]:

```
- : shape = ColorPoint ({x = 0; y = 0}, Green)
```

Handling the exception

Given a list of shapes return `Some p` where `p` is a green point. Otherwise, return `None`.

In [23]:

```
let find_green_point_opt l =
  try Some (find_green_point l) with
  | NoGreenPoint -> None
```

Out[23]:

```
val find_green_point_opt : shape list -> shape option = <fu
n>
```

In [24]:

```
find_green_point_opt []
```

Out[24]:

```
- : shape option = None
```

In [25]:

```
find_green_point_opt [Rect ({x=1;y=1},{x=2;y=2}); ColorPoint ({x=0;y=0},
```

Out[25]:

```
- : shape option = Some (ColorPoint ({x = 0; y = 0}, Gree
n))
```

Exceptions: Recommendations

- Avoid exceptions in your code.
  - Unhandled exceptions are runtime errors; aim to avoid this.
- No exhaustiveness check for exceptions (why?).
- Whenever you might need to use exceptions, think whether you can replace that with

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

or

```
type ('a,'b) result = Ok of 'a | Error of 'b
```

Exercise

`List.hd : 'a list -> 'a` and `List.tl: 'a list -> 'a list` are functions from the list standard library (https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html). They raise exception when the given list is empty. Implement safe versions of the functions whose

signatures are:

In [26]:

```
let safe_hd (l : 'a list) : 'a option = failwith "not implemented"
let safe_tl (l : 'a list) : 'a list option = failwith "not implemented"
```

Out[26]:

```
val safe_hd : 'a list -> 'a option = <fun>
```

Out[26]:

```
val safe_tl : 'a list -> 'a list option = <fun>
```

Fin.