CSE216
Foundations of Computer Science

Instructor: Zhoulai Fu

State University of New York, Korea

Many slides taken from Cornell's CS3110. Thanks!
https://www.cs.cornell.edu/courses/cs3110/2014fa/lecture_notes.php

A review exercise on lists

Exercise #5, #6

- OCaml standard library has List.rev. Reimplement it.
- Find out whether a list is a palindrome. A palindrome is the same as its own reverse, e.g. ['s','m','i','m','s']

Many good exercises taken from “99 Problems in OCaml”

https://v2.ocaml.org/learn/tutorials/99problems.html

Option

Question

- How would you implement maximum of a list?

```
let rec max_list (lst : int list) : int =
  match lst with
    [] -> ???
  | h::t -> max(h,max_list(t))
```

Ocaml likes to use Option for such a situation

Type *t option*

- A value v has type **t option** if it is either:
  - – the value **None**, or
  - – a value **Some** v, and v has type t
  - type 'a option = None | Some of 'a
- Options can signal there is no useful result to the computation
  - • Example: we loop up a value in a hash table using a key. – If the key is present in the hash table then we return Some v where v is the associated value
  - – If the key is not present, we return None

Constructing an option

- None
- Some 1
- Some “hi”

Accessing an option

```
match e with
    None -> ...
  | Some x -> ...
```

Revisit: What is max of empty list?

```
let max (x, y) =
  if x>y then x else y

let rec max_list (lst : int list) : int option =
  match lst with
    []     -> None
  | h::t -> match max_list(t) with
              None    -> Some h
            | Some x -> Some (max(h,x))
```

Very stylish!
...no possibility of exceptions
...no chance of programmer ignoring a "null return"

Exercise #1

- Write a function **last : 'a list -> 'a option** that returns the last element of a list.

```
# last ["a" ; "b" ; "c" ; "d"];;
- : string option = Some "d"
# last [];;
- : 'a option = None
```

Exercise #2

- Find the last two elements of a list.

```
# last_two ["a"; "b"; "c"; "d"];;
- : (string * string) option = Some ("c", "d")
# last_two ["a"];;
- : (string * string) option = None
```

Exercise #3

- Find the K'th element of a list.

```
# at 3 ["a"; "b"; "c"; "d"; "e"];;
- : string option = Some "c"
# at 3 ["a"];;
- : string option = None
```

ALGEBRAIC DATATYPES

Recall: datatype for days

```
type day = Sun | Mon | Tue | Wed
         | Thu | Fri | Sat
```

One-of type
Each "branch" is a *constructor*

Recall: datatype for option

- type 'a option = None | Some of 'a
- Some can carry data

Algebraic datatypes

```
type mytype = TwoInts of int * int
            | Str of string
            | Pizza
```

- Each constructor can *carry* data along with it
- A constructor behaves like a function that makes values of the new type (or is a value of the new type):
  - `TwoInts : int * int -> mytype`
  - `Str : string -> mytype`
  - `Pizza : mytype`

Algebraic datatypes (2)

```
type mytype = TwoInts of int * int
            | Str of string
            | Pizza
```

- Any value of type **`mytype`** is made from *one of* the constructors
- The value contains:
  - A "tag" for "which constructor" (e.g., **`TwoInts`**)
  - The corresponding data (e.g., **`(7,9)`**)

Accessing datatypes values

- So we know how to build datatype values; need to access them
- There are two aspects to accessing a datatype value
- 1. Check what variant it is (what constructor made it)
- 2. Extract the data (if that variant carries any)

Accessing datatypes values with Pattern matching

```
let f (x:mytype) : int =
  match x with
    Pizza -> 3
  | TwoInts(i1,i2) -> i1+i2
  | Str s -> String.length s
```

- One branch per variant
- Each branch
  - extracts the carried data and
  - binds data to variables local to that branch

Pattern matching algebraic datatypes — summary

Syntax:

```
match e0 with
  p1 -> e1
| p2 -> e2
| ...
| pn -> en
```

For now, each *pattern* is a constructor name followed by the right number of variables (i.e., `C` or `C x` or `C(x,y)` or …)

- Syntactically patterns might look like expressions
- But patterns are not expressions
  - OCaml does not evaluate patterns
  - OCaml does determine whether result of `e0` *matches* patterns

Why pattern matching is appreciated

- 1. You can’t forget a case (in-exhaustive pattern-match warning)
- 2. You can’t duplicate a case (unused match case warning)
- 3. You can’t get an exception from forgetting to test the variant (e.g., hd [])
- ==> Pattern matching leads to elegant, concise, beautiful code

Summary

- Lists
- Option
- Algebraic datatype
- Pattern matching with Algebraic datatype

Exercise

- Define a type “point”, which is a 2 dimensional point of two floats
- Define a type “shape”, which is a point, a circle, or a rectangle. A circle is a point and a float of radius; a rectangle are two points
- Define a function area : shape -> float
- Define a function center: shape -> point

Solution

```
type point = float * float
type shape =
  | Point of point
  | Circle of point * float (* center and radius *)
  | Rect of point * point (* lower-left and upper-right corners *)

let area = function
  | Point _ -> 0.0
  | Circle (_, r) -> Float.pi *. (r ** 2.0)
  | Rect ((x1, y1), (x2, y2)) ->
      let w = x2 -. x1 in
      let h = y2 -. y1 in
      w *. h

let center = function
  | Point p -> p
  | Circle (p, _) -> p
  | Rect ((x1, y1), (x2, y2)) -> ((x2 +. x1) /. 2.0, (y2 +. y1) /. 2.0)

type point = float * float
type shape = Point of point | Circle of point * float | Rect of point * point
val area : shape -> float = <fun>
val center : shape -> point = <fun>
```