CMSC 330: Organization of Programming Languages

Functional Programming with OCaml, con't.

More Examples

```
(* convert a list of pairs to a list of all the
   elements in the pairs *)
flatten_pairs l  (* ('a * 'a) list -> 'a list *)
   let rec flatten_pairs l = match l with
     [] -> []
    | ((a, b)::t) -> a :: b :: (flatten_pairs t)

(* return a list with the first n elements of l *)
take (n, l)
   let rec take (n, l) =
     if n = 0
       then []
       else
         match l with
           [] -> []
          | (h::t) -> h :: (take (n-1, t))
```


OCaml Data

- So far, we've seen the following kinds of data:
  - basic types (int, float, char, string)
  - lists
    - one kind of data structure
    - a list is either [] or h::t, deconstructed with pattern matching
  - tuples
    - let you collect data together in fixed-size pieces
  - functions
- How can we build other data structures?
  - building everything from lists and tuples is awkward

Data Types

```
type shape =
    Rect of float * float    (* width * length *)
  | Circle of float          (* radius *)

let area s =
  match s with
      Rect (w, l) -> w *. l
    | Circle r -> r  *. r *. 3.14

area (Rect (3.0, 4.0))
area (Circle 3.0)
```

- **`Rect`** and **`Circle`** are *type constructors*- here a **`shape`** is either a **`Rect`** or a **`Circle`**
- Use pattern matching to *deconstruct* values, and do different things depending on constructor

Data Types, con't.

```
type shape =
    Rect of float * float   (* width * length *)
  | Circle of float

let l = [Rect (3.0, 4.0) ; Circle 3.0; Rect (10.0, 22.5)]
```

- What's the type of `l`?
- What's the type of `l`'s first element?

Data Types (cont'd)

- The *arity* of a constructor is the number of arguments it takes
  - a constructor with no arguments is *nullary*

```
type optional_int =
   None
 | Nbr of int

let add_with_default (a, o) =
  match o with
     None -> a + 1
   | Nbr n -> a + n

add_with_default (3, None)      (* 4 *)
add_with_default (3, (Nbr 4))   (* 7 *)
```

  - constructors must begin with uppercase letter

Polymorphic Data Types

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

let add_with_default (a, o) =
  match o with
      None -> a + 1
    | Nbr n -> a + n

add_with_default (3, None)      (* 4 *)
add_with_default (3, (Nbr 4))   (* 7 *)
```

- This option type can work with any kind of data
  - In fact, this option type is built-in to OCaml

Recursive Data Types

- Do you get the feeling we can build up lists this way?

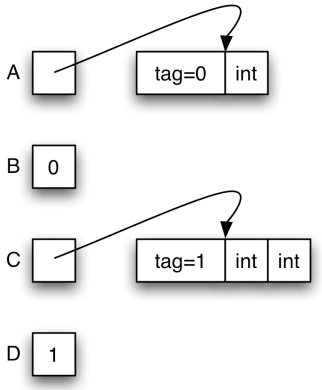
```
type 'a mylist =
   Nil
 | Node of 'a * 'a mylist

let rec length l =
  match l with
    Nil -> 0
  | Node (_, t) -> 1 + (length t)

length (Node (10, Node (20, Node (30, Nil))))
```

  - Note: there's no nice [1; 2; 3] syntax for this kind of list

Data Type Representations

- Values in a data type are stored either directly as integers or as pointers to blocks in the heap

```
type t =
  A of int
| B
| C of int * int
| D
```



Exceptions

- Exceptions are declared with exception
  - They may appear in the signature as well
- Exceptions may take arguments
  - Just like type constructors
  - May also be nullary
- Catch exceptions with try...with...
  - Pattern-matching can be used in with
  - If an exception is uncaught, the current function exits immediately and control transfers up the call chain until the exception is caught, or until it reaches the top level

Exceptions, con't.

```
exception My_exception of int

let f m =
  if m > 0 then
    raise (My_exception m)
  else
    raise (Failure "some explanation of failure type")

let g n =
  try
    f n
  with My_exception n ->
      Printf.printf "Caught %d\n" n
    | Failure s ->
      Printf.printf "Caught %s\n" s
```


Working with Lists

- Several of the recursive list function examples have the same flavor:
  - walk through a list and do something to every element
  - walk through a list and keep track of something
- Recall the following example code from Ruby:

```
a = [1,2,3,4,5]
b= a.collect { |x| -x }
```

  - here we passed a code block into the collect method
  - wouldn't it be nice to do the same in OCaml?

Higher-Order Functions

- In OCaml you can pass functions as arguments, and return functions as results

```
let plus_three x = x + 3
let twice (f, z) = f (f z)
twice (plus_three, 5)
twice : ('a->'a) * 'a -> 'a

let plus_four x = x + 4
let pick_fn n =
    if n > 0 then plus_three else plus_four
(pick_fn 5) 0
pick_fn : int -> (int -> int)
```