OCaml Data Types
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 and Records**
    - 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
User Defined Types

- **type** can be used to create new names for types

- Like **typedef** in C – a name might be more useful for communicating intent than just the type structure
User Defined Types

```ocaml
# type mylist = int*(int list);;
type mylist = int * int list
# let empty:mylist = (0,[]);;
val empty : mylist = (0, [])
# let add x ((n,xs):mylist):mylist = (n+1,x::xs);;
val add : int -> mylist -> mylist = <fun>
# let length ((n,_):mylist) = n;;
val length : mylist -> int = <fun>
# let x = add 1 (add 2 empty);;
val x : mylist = (2, [1; 2])
```

Annotation required to tell type inference you want mylist, not int*int list.
(User-Defined) Variants

type coin = Heads | Tails

let flip x =
    match x with
    | Heads -> Tails
    | Tails -> Heads

let rec count_heads x =
    match x with
    | [] -> 0
    | (Heads::x') -> 1 + count_heads x'
    | (_::x') -> count_heads x'

In simplest form:
Like a C enum

Basic pattern
matching
resembles C
switch

Combined list
and variant
patterns
possible
Constructing and Destructing Variants

• **Syntax**
  - **type** \( t = C_1 \mid \ldots \mid C_n \)
  - the \( C_i \) are called **constructors**
    - Must begin with a capital letter

• **Evaluation**
  - A constructor \( C_i \) is already a value
  - Destructing a value \( v \) of type \( t \) is done by pattern matching on \( v \); the patterns are the constructors \( C_i \)

• **Type Checking**
  - \( C_i : t \) (for each \( C_i \) in \( t \)'s definition)
Data Types: Variants with Data

- We can define variants that “carry data” too
  - Not just a constructor, but a constructor *with values*

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

- *Rect* and *Circle* are constructors, so a *shape* is either
  - *Rect*(\(w, l\)) for any floats \(w\) and \(l\), or
  - *Circle* \(r\) for any float \(r\)
Data Types: Pattern Matching

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

area (Rect (3.0, 4.0));; (* 12.0 *)
area (Circle 3.0);; (* 28.26 *)

- Use pattern matching to deconstruct values
  - Can bind pattern values to data parts

Data types are aka algebraic data types and tagged unions
Data Types: Pattern Matching

```ocaml
let lst = [Rect (3.0, 4.0); Circle 3.0]
```

- What's the type of `lst`?
  - `shape list`

- What's the type of `lst`'s first element?
  - `shape`
Quiz 1

type foo = (int * (string list)) list

Which one of the following could match type foo?

A. [(3, "foo", "bar")]
B. [(7, ["foo", "bar"])]
C. [(5, ["foo"; "bar"])]
D. [(9, [(["foo", "bar"])])]

Quiz 1

type foo = (int * (string list)) list

Which one of the following could match type foo?

A. [[(3, "foo", "bar")]]
B. [[[7, ["foo", "bar"]]]]
C. [[[5, ["foo"; "bar"]]]]
D. [[[9, [(["foo", "bar"])]]]]
Quiz 2: What does this evaluate to?

type num = Int of int | Float of float;;
let aux a =
  match a with
  | Int i -> float_of_int i
  | Float j -> j
;;
aux (Int 2);;

A. 4.0
B. 2.0
C. 2
D. Type Error
Quiz 2: What does this evaluate to?

```
type num = Int of int | Float of float;;
let aux a =
  match a with
  | Int i -> float_of_int i
  | Float j -> j
;;
aux (Int 2);
```

A. 4.0
B. 2.0
C. 2
D. Type Error
public interface Shape {
    public double area();
}

class Rect implements Shape {
    private double width, length;

    Rect (double w, double l) {
        this.width = w;
        this.length = l;
    }

double area() {
    return width * length;
}
}

class Circle implements Shape {
    private double rad;

    Circle (double r) {
        this.rad = r;
    }

double area() {
    return rad * rad * 3.14159;
}
}
Option Type

- Comparing to Java: **None** is like **null**, while **Some i** is like an **Integer(i)** object

```ocaml
type optional_int = None | Some of int
let divide x y = if y != 0 then Some (x/y) else None
let string_of_opt o = match o with
        Some i -> string_of_int i
    | None -> "nothing"

let p = divide 1 0;;
print_string (string_of_opt p);;
(* prints "nothing" *)
let q = divide 1 1;;
print_string (string_of_opt q);;
(* prints "1" *)
```
Polymorphic Option Type

- A Polymorphic version of option type can work with any kind of data
  - As int option, char option, etc...

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

let p = opthd [];; (* p = None *)
let q = opthd [1;2];; (* q = Some 1 *)
let r = opthd ["a"];; (* r = Some "a" *)
```

In fact, this option type is built into OCaml.

Polymorphic parameter: like Option<T> in Java

```ocaml
let opthd l =
  match l with
  | [] -> None
  | x::_ -> Some x
```
Quiz 3: What does this evaluate to?

\[
\text{let } \text{foo } f = \text{match } f \text{ with } \\
\phantom{\text{let foo } f = \text{match } f \text{ with } } \text{None } \rightarrow 42.0 \\
\phantom{\text{let foo } f = \text{match } f \text{ with } } | \text{Some } n \rightarrow n +. 42.0 \\
\phantom{\text{let foo } f = \text{match } f \text{ with } } ;; \\
\text{foo } 3.3;;
\]

A. 45.3
B. 42.0
C. Some 45.3
D. Error
Quiz 3: What does this evaluate to?

```
let foo f = match f with
    None -> 42.0
  | Some n -> n +. 42.0

foo 3.3
```

A. 45.3
B. 42.0
C. Some 45.3
D. Error
Recursive Data Types

- We can build up lists with **recursive** variant types

```ocaml
type 'a mylist =
  Nil
 | Cons of 'a * 'a mylist

let rec len x = match x with
  Nil -> 0
 | Cons (_, t) -> 1 + (len t)

len (Cons (10, Cons (20, Cons (30, Nil))))
(* evaluates to 3 *)
```

- Won’t have nice [1; 2; 3] syntax for this kind of list
Variants (full definition)

- **Syntax**
  - \texttt{type } \textit{t} = \textit{C1} [of \textit{t1}] \mid \ldots \mid \textit{Cn} [of \textit{tn}]
  - the \textit{Ci} are called constructors
    - Must begin with a capital letter; may include associated data - notated with brackets \texttt{[]} to indicate it’s optional

- **Evaluation**
  - A constructor \textit{Ci} is a value if it has no assoc. data
    - \textit{Ci vi} is a value if it does
  - Destructing a value of type \textit{t} is by pattern matching
    - patterns are constructors \textit{Ci} with data components, if any

- **Type Checking**
  - \texttt{Ci [vi] : t [if vi has type ti]}
exception My_exception of int
let f n =
  if n > 0 then
    raise (My_exception n)
  else
    raise (Failure "foo")
let bar n =
try
  f n
with My_exception n ->
  Printf.printf "Caught %d\n" n
| Failure s ->
  Printf.printf "Caught %s\n" s
OCaml Exceptions: Details

- Exceptions are declared with `exception`
  - They may appear in the signature as well
- Exceptions may take arguments
  - Just like type constructors
  - May also have no arguments
- Catch exceptions with `try...with...`
  - Pattern-matching can be used in `with`
  - If an exception is uncaught
    - Current function exits immediately
    - Control transfers up the call chain
    - Until the exception is caught, or until it reaches the top level
OCaml Exceptions: Useful Examples

- **failwith s**: Raises exception Failure s (s is a string).
- **Not_found**: Exception raised by library functions if the object does not exist
- **invalid_arg s**: Raises exception Invalid_argument s

```ocaml
let div x y =  
  if y = 0 then failwith "div by 0" else x/y;;

let lst =[(1,"alice");(2,"bob");(3,"cat")];;
let lookup key lst =
  try
    List.assoc key lst
  with
    Not_found -> "key does not exist"
```