OCaml Datatypes

COS 326
David Walker
Princeton University
OCaml So Far

• We have seen a number of basic types:
  – int
  – float
  – char
  – string
  – bool

• We have seen a few structured types:
  – pairs
  – tuples
  – options
  – lists

• In this lecture, we will see some more general ways to define our own new types and data structures
Type Abbreviations

- We have already seen some type abbreviations:

```plaintext
type point = float * float
```
Type Abbreviations

• We have already seen some type abbreviations:

```plaintext
type point = float * float
```

• These abbreviations can be helpful documentation:

```plaintext
let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

• But they add nothing of *substance* to the language
  – they are *equal* in every way to an existing type
Type Abbreviations

• We have already seen some type abbreviations:

\[
\text{type point} = \text{float} \times \text{float}
\]

• As far as OCaml is concerned, you could have written:

```ocaml
let distance (p1: float*float) (p2: float*float) : float =
    let square x = x *. x in
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

• Since the types are equal, you can substitute the definition for the name wherever you want
  – we have not added any new data structures
DATA TYPES
OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives.

```ocaml
type my_bool = Tru | Fal
```

A value with type `my_bool` is one of two things:
- `Tru`, or
- `Fal`

Read the `|` as "or".
OCaml provides a general mechanism called a **data type** for defining new data structures that consist of many alternatives.

```ocaml
type my_bool = Tru | Fal
```

A value with type `my_bool` is one of two things:
- `Tru`, or
- `Fal`

Tru and Fal are called "constructors".

Read the "|" as "or".
OCaml provides a general mechanism called a **data type** for defining new data structures that consist of many alternatives.

```ocaml
type my_bool = True | False

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

There's no need to stop at 2 cases; define as many alternatives as you want.
Data types

• OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```ocaml
type my_bool = Tru | Fal

let b1 : my_bool = Tru
let b2 : my_bool = Fal
let c1 : color = Yellow
let c2 : color = Red
```

• Creating values:

use constructors to create values
Data types

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

let c1 : color = Yellow
let c2 : color = Red
```

- **Using data type values:**

```ocaml
let print_color (c:color) : unit =
  match c with
  | Blue ->
  | Yellow ->
  | Green ->
  | Red ->
```

use pattern matching to determine which color you have; act accordingly
Data types

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

let c1 : color = Yellow
let c2 : color = Red

• Using data type values:

let print_color (c:color) : unit =
    match c with
    | Blue -> print_string "blue"
    | Yellow -> print_string "yellow"
    | Green -> print_string "green"
    | Red -> print_string "red"
```
• Using data type values:

```haskell
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Green -> print_string "green"
  | Red -> print_string "red"
```

Why not just use strings to represent colors instead of defining a new type?
Data types

type color = Blue | Yellow | Green | Red

oops!:

let print_color (c:color) : unit =
    match c with
    | Blue -> print_string "blue"
    | Yellow -> print_string "yellow"
    | Red -> print_string "red"

Warning 8: this pattern-matching is not exhaustive. Here is an example of a value that is not matched: Green
Data types

```ocaml
let print_color (c:color) : unit =
    match c with
    | Blue -> print_string "blue"
    | Yellow -> print_string "yellow"
    | Red -> print_string "red"
```

Warning 8: this pattern-matching is not exhaustive. Here is an example of a value that is not matched: Green

OCaml's datatype mechanism allow you to create types that contain precisely the values you want!
Data Types Can Carry Additional Values

- Data types are more than just enumerations of constants:

```haskell
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float
```

- Read as: a `simple_shape` is either:
  - a `Circle`, which contains a pair of a `point` and `float`, or
  - a `Square`, which contains a pair of a `point` and `float`
Data Types Can Carry Additional Values

- Data types are more than just enumerations of constants:

```ocaml
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let origin : point = (0.0, 0.0)

let circ1  : simple_shape = Circle (origin, 1.0)
let circ2  : simple_shape = Circle ((1.0, 1.0), 5.0)
let square : simple_shape = Square (origin, 2.3)
```
Data Types Can Carry Additional Values

- Data types are more than just enumerations of constants:

```plaintext
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let simple_area (s:simple_shape) : float =
  match s with
  | Circle (_, radius) -> 3.14 *. radius *. radius
  | Square (_, side) -> side *. side
```
Compare

- Data types are more than just enumerations of constants:

```ocaml
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let simple_area (s:simple_shape) : float =
  match s with
  | Circle (_, radius) -> 3.14 *. radius *. radius
  | Square (_, side) -> side *. side

let simple_area (s:my_shape) : float =
  (3.14 *. radius *. radius) ?? or ?? (side *. side)
```
More General Shapes

```ocaml
type point = float * float

type shape =
  Square of float
| Ellipse of float * float
| RtTriangle of float * float
| Polygon of point list
```

Square \( s = \)

RtTriangle \( (s_1, s_2) = \)

Ellipse \( (r_1, r_2) = \)

Polygon \( [v_1; \ldots; v_5] = \)
type point = float * float

type radius = float

type side = float

type shape =
    Square of side
    | Ellipse of radius * radius
    | RtTriangle of side * side
    | Polygon of point list

Type abbreviations can aid readability

Square $s = \begin{array}{c}
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet}
\end{array}$ $s$

RtTriangle $(s_1, s_2) = \triangle s_1 s_2$

Ellipse $(r_1, r_2) = \text{\textbullet} r_1 r_2$

RtTriangle $[v_1; ...; v_5] =$
type point = float * float
type radius = float
type side = float

type shape =
  Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list

let sq  : shape = Square 17.0
let ell : shape = Ellipse (1.0, 2.0)
let rt  : shape = RtTriangle (1.0, 1.0)
let poly : shape = Polygon [(0., 0.); (1., 0.); (0.; 1.)]

they are all shapes; they are constructed in different ways

Square builds a shape from a single side

RtTriangle builds a shape from a pair of sides

Polygon builds a shape from a list of points (where each point is itself a pair)
More General Shapes

```ocaml
type point = float * float
type radius = float
type side = float

type shape =
    Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list

let area (s : shape) : float =
  match s with
  | Square s ->
  | Ellipse (r1, r2) ->
  | RtTriangle (s1, s2) ->
  | Polygon ps ->
```

A data type also defines a pattern for matching.
type point = float * float
type radius = float
type side = float

type shape =
    Square of side
| Ellipse of radius * radius
| RtTriangle of side * side
| Polygon of point list

let area (s : shape) : float =
    match s with
    | Square s ->
    | Ellipse (r1, r2) ->
    | RtTriangle (s1, s2) ->
    | Polygon ps ->

Square carries a value with type float so s is a pattern for float values

RtTriangle carries a value with type float * float so (s1, s2) is a pattern for that type

a data type also defines a pattern for matching
More General Shapes

type point = float * float

type radius = float

type side = float

type shape =
  Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list

let area (s : shape) : float =
  match s with
  | Square s -> s *. s
  | Ellipse (r1, r2)-> pi *. r1 *. r2
  | RtTriangle (s1, s2) -> s1 *. s2 / 2.
  | Polygon ps -> ???

a data type also defines a pattern for matching
How do we compute polygon area?

For convex polygons:

- Case: the polygon has fewer than 3 points:
  - it has 0 area! (it is a line or a point or nothing at all)

- Case: the polygon has 3 or more points:
  - Compute the area of the triangle formed by the first 3 vertices
  - Delete the second vertex to form a new polygon
  - Sum the area of the triangle and the new polygon

\[ \text{v2} \quad \text{v1} \quad \text{v3} \quad \text{v4} \quad \text{v5} = \text{triangle} + \text{new polygon} \]
Computing Area

• How do we compute polygon area?
• For convex polygons:
  – Case: the polygon has fewer than 3 points:
    • it has 0 area! (it is a line or a point or nothing at all)
  – Case: the polygon has 3 or more points:
    • Compute the area of the triangle formed by the first 3 vertices
    • Delete the second vertex to form a new polygon
    • Sum the area of the triangle and the new polygon
• Note: This is a beautiful inductive algorithm:
  – the area of a polygon with \( n \) points is computed in terms of a smaller polygon with only \( n-1 \) points!
let area (s : shape) : float =
    match s with
    | Square s -> s *. s
    | Ellipse (r1, r2) -> r1 *. r2
    | RtTriangle (s1, s2) -> s1 *. s2 / 2.
    | Polygon ps -> poly_area ps

let poly_area (ps : point list) : float =
    match ps with
    | p1 :: p2 :: p3 :: tail ->
        tri_area p1 p2 p3 +. poly_area (p1::p3::tail)
    | _ -> 0.

This pattern says the list has at least 3 items
let tri_area (p1:point) (p2:point) (p3:point) : float = 
  let a = distance p1 p2 in 
  let b = distance p2 p3 in 
  let c = distance p3 p1 in 
  let s = 0.5 *. (a +. b +. c) in 
  sqrt (s *. (s -. a) *. (s -. b) *. (s -. c)) 

let rec poly_area (ps : point list) : float = 
  match ps with 
  | p1 :: p2 :: p3 :: tail ->
    tri_area p1 p2 p3 +. poly_area (p1::p3::tail)
  | _ -> 0.

let area (s : shape) : float = 
  match s with 
  | Square s -> s *. s 
  | Ellipse (r1, r2)-> pi *. r1 *. r2 
  | RtTriangle (s1, s2) -> s1 *. s2 /. 2.
  | Polygon ps -> poly_area ps
INDUCTIVE DATA TYPES
Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a **Leaf** containing no data
  - a **Node** containing a **key**, a **value**, a left subtree and a right subtree
Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree

```plaintext
type key = string
type value = int

type tree =
    Leaf
  | Node of key * value * tree * tree
```
Inductive data types

type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree

let rec insert (t:tree) (k:key) (v:value) : tree =

type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree

let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf ->
  | Node (k', v', left, right) ->
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree

let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
Inductive data types

type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree

let rec insert (t:tree) (k:key) (v:value) : tree =
match t with
    | Leaf -> Node (k, v, Leaf, Leaf)
    | Node (k', v', left, right) ->
        if k < k' then
            Node (k', v', insert left k v, right)
        else if k > k' then
            Node (k', v', left, insert right k v)
        else
            Node (k, v, left, right)
Inductive data types

type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree

let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
    if k < k' then
      Node (k', v', insert left k v, right)
    else if k > k' then
      Node (k', v', left, insert right k v)
    else
      Node (k, v, left, right)
Inductive data types

type key = int
value = string

type tree =
  Leaf
| Node of key * value * tree * tree

let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
    if k < k' then
      Node (k', v', insert left k v, right)
    else if k > k' then
      Node (k', v', left, insert right k v)
    else
      Node (k, v, left, right)
Recall, we used the type "int" to represent natural numbers
– but that was kind of broken: it also contained negative numbers
– we had to use a dynamic test to guard entry to a function:

```ocaml
let double (n : int) : int =
  if n < 0 then raise (Failure "negative input!")
  else double_nat n
```

– it would be nice if there was a way to define the natural numbers exactly, and use OCaml's type system to guarantee no client ever attempts to double a negative number
Inductive data types

• Recall, a natural number $n$ is either:
  – zero, or
  – $m + 1$

• We use a data type to represent this definition exactly:
Inductive data types

• Recall, a natural number \( n \) is either:
  – zero, or
  – \( m + 1 \)

• We use a data type to represent this definition exactly:

\[
\text{type nat} = \text{Zero} \mid \text{Succ of nat}
\]
Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

```ocaml
type nat = Zero | Succ of nat

let rec nat_to_int (n : nat) : int =
  match n with
  Zero -> 0
  | Succ n -> 1 + nat_to_int n
```
Inductive data types

• Recall, a natural number \( n \) is either:
  – zero, or
  – \( m + 1 \)

• We use a data type to represent this definition exactly:

```ocaml
type nat = Zero | Succ of nat

let rec nat_to_int (n : nat) : int =
  match n with
  Zero -> 0
| Succ n -> 1 + nat_to_int n

let rec double_nat (n : nat) : nat =
  match n with
  Zero -> Zero
| Succ m -> Succ (Succ(double_nat m))
```
Summary

• OCaml data types: a powerful mechanism for defining complex data structures:
  – They are precise
    • contain exactly the elements you want, not more elements
  – They are general
    • recursive, non-recursive (mutually recursive and polymorphic)
  – The type checker helps you detect errors
    • missing cases in your functions