OCaml Datatypes

COS 326
Andrew Appel
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:

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

• These abbreviations can be helpful documentation:

```ocaml
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:

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

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

```
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
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,
- 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
```

Tru and Fal are called "constructors".

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

Read the "|" as "or".
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

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.

- Creating values:

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

Use constructors to create values.
Data types

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

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

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

- **Using data type values:**

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

```
-- Using data type values:

type color = Blue | Yellow | Green | Red

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

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"
```
Data types

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

  let c1 : color = Yellow
  let c2 : color = 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

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

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:

```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
```
Compare

- 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

let simple_area (s:my_shape) : float =
    (3.14 *. radius *. radius) ?? or ?? (side *. side)
```
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 \( [v_1; \ldots; v_5] = \)
**More General Shapes**

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

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

Type abbreviations can aid readability

Square \( s = \)

RtTriangle \( (s1, s2) = \)

Ellipse \( (r1, r2) = \)

RtTriangle \( [v1; ...;v5] = \)
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 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.)]
```

- 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)

They are all shapes; they are constructed in different ways.
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
More General Shapes

```ml
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

```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 -> 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.
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

\[ v_1 v_2 v_3 = \triangle + \text{new polygon} \]
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!
Computing Area

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

```ocaml
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 =
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', left, right) ->

Again, the type definition specifies the cases you must consider
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) ->
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

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: Another Example

- 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:

```haskell
type nat = Zero | 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 m -> 1 + nat_to_int n

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

- Recall, a list is either:
  - nil, or
  - the cons of a *head* value with a *tail* list

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

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

• 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
OCaml Datatypes Part II: An Exercise in Type Design
IBM developed GML (Generalize Markup Language) in 1969

- Precursor to SGML, HTML and XML

Chapter 1: Introduction

GML supported hierarchical containers, such as ordered lists (like this one), unordered lists, and definition lists as well as simple structures.

Markup Minimization (later generalized and formalized in SGML), allowed the end-tags to be omitted for the “h1” and “p” elements.
To process a GML document, an OCaml program would:

• **Read** a series of characters from a text file & **Parse** GML structure
• **Represent** the information content as an OCaml data structure
• **Analyze** or **transform** the data structure
• **Print/Store/Communicate** results

We will focus on how to **represent** and **transform** the information content of a GML document.
• A GML document consists of:
  – a list of elements

• An element is either:
  – a word or markup applied to an element

• Markup is either:
  – italicize, bold, or a font name
A GML document consists of:
  - a list of elements

An element is either:
  - a word or markup applied to an element

Markup is either:
  - italicize, bold, or a font name
**Example Data**

```ocaml
let d = [ Formatted (Bold,
   Formatted (Font "Arial",
    Words ["Chapter";"One"]));

Words ["It"; "was"; "a"; "dark";
   "&"; "stormy; "night."; "A"];

Formatted (Ital, Words["shot"]);

Words ["rang"; "out."] ];;;
```
Challenge

• Change all of the “Arial” fonts in a document to “Courier”.

• Of course, when we program functionally, we implement \textit{change} via a function that
  – receives one data structure as input
  – builds a new (different) data structure as an output
Challenge

• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```
• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
let rec chfonts (elts:doc) : doc =
```

• Technique: approach the problem top down, work on `doc` first:
Challenge

• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
    Words of string list
  | Formatted of markup * elt

type doc = elt list

let rec chfonts (elts:doc) : doc =
  match elts with
  | [] ->
  | hd::tl ->
```
Challenge

• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

• Technique: approach the problem top down, work on doc first:

```ocaml
let rec chfonts (elts:doc) : doc =
  match elts with
  | [] -> []
  | hd::tl -> (chfont hd)::(chfonts tl)
```
• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

• Next work on changing the font of an element:

```ocaml
let rec chfont (e:elt) : elt =
```
Changing fonts in an element

• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

• Next work on changing the font of an element:

```ocaml
let rec chfont (e:elt) : elt =
  match e with
  | Words ws ->
  | Formatted(m,e) ->
```
Changing fonts in an element

• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

• Next work on changing the font of an element:

```ocaml
let rec chfont (e:elt) : elt =
  match e with
  | Words ws -> Words ws
  | Formatted(m,e) ->
```
Changing fonts in an element

• Change all of the “Arial” fonts in a document to “Courier”.

```
type markup = Ital | Bold | Font of string

type elt =
    Words of string list
| Formatted of markup * elt

type doc = elt list
```

• Next work on changing the font of an element:

```
let rec chfont (e:elt) : elt =
    match e with
    | Words ws -> Words ws
    | Formatted(m,e) -> Formatted(chmarkup m, chfont e)
```
Changing fonts in an element

• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

• Next work on changing a markup:

```ocaml
let chmarkup (m:markup) : markup =
```
Changing fonts in an element

• Change all of the “Arial” fonts in a document to “Courier”.

```ocaml
type markup = Ital | Bold | Font of string

type elt =
    Words of string list
| Formatted of markup * elt

type doc = elt list
```

• Next work on changing a markup:

```ocaml
let chmarkup (m:markup) : markup =
    match m with
    | Font “Arial” -> Font “Courier”
    | _ -> m
```
Summary: Changing fonts in an element

• Change all of the “Arial” fonts in a document to “Courier”
• Lesson: function structure follows type structure

```ocaml
let chmarkup (m:markup) : markup =
    match m with
    | Font "Arial" -> Font "Courier"
    | _ -> m

let rec chfont (e:elt) : elt =
    match e with
    | Words ws -> Words ws
    | Formatted(m,e) -> Formatted(chmarkup m, chfont e)

let rec chfonts (elts:doc) : doc =
    match elts with
    | [] -> []
    | hd::tl -> (chfont hd)::(chfonts tl)
```
• Consider again our definition of markup and markup change:

```csharp
type markup =
    Ital | Bold | Font of string

let chmarkup (m:markup) : markup =
    match m with
    | Font "Arial" -> Font "Courier"
    | _ -> m
```
• What if we make a change:

```ocaml
type markup =
  Ital | Bold | Font of string | TTFont of string

let chmarkup (m:markup) : markup =
  match m with
  | Font "Arial" -> Font "Courier"
  | _ -> m
```

the underscore silently catches all possible alternatives

this may not be what we want -- perhaps there is an Arial TT font

it is better if we are alerted of all functions whose implementation may need to change
Better Style

• Original code:

```ocaml
type markup =
  Ital | Bold | Font of string

let chmarkup (m:markup) : markup =
match m with
| Font "Arial" -> Font "Courier"
| Ital | Bold -> m
```
• Updated code:

```haskell
type markup =
    Ital | Bold | Font of string | TTFont of string

let chmarkup (m:markup) : markup =
    match m with
    | Font "Arial" -> Font "Courier"
    | Ital | Bold -> m

..match m with
    | Font "Arial" -> Font "Courier"
    | Ital | Bold -> m..

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
TTFont _
```
• Updated code, fixed:

```ocaml
type markup =
  Ital | Bold | Font of string | TTFONT of string

let chmarkup (m:markup) : markup =
  match m with
  | Font "Arial" -> Font "Courier"
  | TTFONT "Arial" -> TTFONT "Courier"
  | Font s -> Font s
  | TTFONT s -> TTFONT s
  | Ital | Bold -> m
```

• **Lesson**: use the type checker where possible to help you maintain your code
A couple of practice problems

• Write a function that gets rid of immediately redundant markup in a document.
  – `Formatted(Ital, Formatted(Ital,e))` can be simplified to `Formatted(Ital,e)`
  – write maps and folds over markups

• Design a datatype to describe bibliography entries for publications. Some publications are journal articles, others are books, and others are conference papers. Journals have a name, number and issue; books have an ISBN number; All of these entries should have a title and author.
  – design a sorting function
  – design maps and folds over your bibliography entries
To Summarize

- Design recipe for writing OCaml code:
  - write down English specifications
    - try to break problem into obvious sub-problems
  - write down some sample test cases
  - write down the signature (types) for the code
  - use the signature to guide construction of the code:
    - tear apart inputs using pattern matching
      - make sure to cover all of the cases! (OCaml will tell you)
    - handle each case, building results using data constructor
      - this is where human intelligence comes into play
      - the “skeleton” given by types can almost be done automatically!
    - clean up your code
  - use your sample tests (and ideally others) to ensure correctness