Data, data everywhere

COS 326
David Walker
Princeton University
FUNCTIONAL DECOMPOSITION
Functional Decomposition

==

Break down complex problems into a set of simple functions;
Recombine (compose) functions to form solution

Such problems can often be solved using a combinator library.
(a set of functions that fit together nicely)

The list library, which contains map and fold, is a combinator library.
PIPELINES
Pipe

let (|>) x f = f x

Type?
let (|>) x f = f x

Type?

(|>) : 'a -> ('a -> 'b) -> 'b
let ( |> ) x f = f x

let twice f x =
  x |> f |> f
let (|>) x f = f x

let twice f x =
  (x |> f) |> f

left associative: x |> f1 |> f2 |> f3 == ((x |> f1) |> f2) |> f3
let (|>) x f = f x

let twice f x =
  x |> f |> f

let square x = x*x

let fourth x = twice square
let (|>) x f = f x

let twice f x = x |> f |> f
let square x = x*x
let fourth x = twice square x

let compute x =
    x |> square
    |> fourth
    |> ( * ) 3
    |> print_int
    |> print_newline
PIPING LIST PROCESSORS
(Combining combinators cleverly)
type student = {first: string; 
    last:   string; 
    assign: float list; 
    final:  float}

let students : student list = 
[
    {first = "Sarah";
    last  = "Jones";
    assign = [7.0;8.0;10.0;9.0];
    final  = 8.5};

    {first = "Qian";
    last  = "Xi";
    assign = [7.3;8.1;3.1;9.0];
    final  = 6.5};
]

Another Problem

```javascript
type student = {first: string;
               last:  string;
               assign: float list;
               final:  float}
```

- Create a function `display` that does the following:
  - for each student, print the following:
    - `last_name, first_name: score`
    - `score` is computed by averaging the assignments with the final
      - each assignment is weighted equally
      - the final counts for twice as much
    - one student printed per line
    - students printed in order of score
Do Professors Dream of Homework-grade Databases?

(1968 novel)
Create a function **display** that

– takes a list of students as an argument
– prints the following for each student:
  
  • last_name, first_name: score
  • score is computed by averaging the assignments with the final
    – each assignment is weighted equally
    – the final counts for twice as much
  • one student printed per line
  • students printed in order of score

```ocaml
let display (students : student list) : unit =
  students |> compute score
  |> sort by score
  |> convert to list of strings
  |> print each string
```
Another Problem

let compute_score {first=f; last=l; assign=grades; final=exam} =

let sum x (num,tot) = (num +. 1., tot +. x) in

let score gs e = List.fold_right sum gs (2., 2. *. e) in

let (number, total) = score grades exam in
(f, l, total /. number)

let display (students : student list) : unit =
students |> List.map compute_score
|> sort by score
|> convert to list of strings
|> print each string
let student_compare (_,_,score1) (_,_,score2) = 
  if score1 < score2 then 1 
  else if score1 > score2 then -1 
  else 0

let display (students : student list) : unit = 
  students |> List.map compute_score 
  |> List.sort compare_score 
  |> convert to list of strings 
  |> print each string
Another Problem

```haskell
let display (students : student list) : unit =
  students |> List.map compute_score
  |> List.sort compare_score
  |> List.map stringify
  |> print each string

let stringify (first, last, score) =
  last ^ "", " ^ first ^ ": " ^ string_of_float score
```
Another Problem

```ocaml
let display (students : student list) : unit =
    students |
    List.map compute_score |
    List.sort compare_score |
    List.map stringify |
    List.iter print_endline
```

```ocaml
let stringify (first, last, score) =
    last ^ ", " ^ first ^ ": " ^ string_of_float score
```
COMBINATORS FOR OTHER TYPES: PAIRS
let both $f (x, y) = (f x, f y);$;
let do_fst $f (x, y) = (f x, y);$;
let do_snd $f (x, y) = (x, f y);;$
Example: Piping Pairs

let both f (x,y) = (f x, f y);;
let do_fst f (x,y) = (f x, y);;
let do_snd f (x,y) = (x, f y);;

let even x = (x/2)*2 == x;;

let process (p : float * float) =
  p |> both int_of_float (* convert to int *)
  |> do_fst ((/ 3)) (* divide fst by 3 *)
  |> do_snd ((/ 2)) (* divide snd by 2 *)
  |> both even (* test for even *)
  |> fun (x,y) -> x && y (* both even *)
When & how to create new combinator libraries?

Whenever you see a need!

Are there specialized programming domains you are familiar with?

Can you identify the repeated patterns in examples?

Can you factor out the repetitions into reuseable fragments?

Do you need to generalize to create uniformity?

Is there data that flows between components?

What types describe such data and form the interfaces?

There is a lot of art, aesthetics, and experience involved. ICFP is conference where you can find many interesting combinator libraries http://www.icfpconference.org/
Summary

• (|>) passes data from one function to the next
  – compact, elegant, clear

• UNIX pipes (|) compose file processors
  – unix scripting with | is a kind of functional programming
  – but it isn't very general since | is not polymorphic
  – you have to serialize and deserialize your data at each step
    • there can be type (ie: file format) mismatches between steps
    • we avoided that in your assignment, which is pretty simple ...

• Higher-order *combinator libraries* arranged around types:
  – List combinators (map, fold, reduce, iter, ...)
  – Pair combinators (both, do_fst, do_snd, ...)
  – Network programming combinators (Frenetic: frenetic-lang.org)
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:

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

- We have already seen some type abbreviations:
  
  \[
  \text{type point} = \text{float} \times \text{float}
  \]

- These abbreviations can be helpful documentation:
  
  ```
  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 \textit{substance} to the language
  – they are \textit{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
DATA TYPES
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"
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
```

- Tru and Fal are called "constructors".
- 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
```

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

There's no need to stop at 2 cases; define as many alternatives as you want.
OCaml provides a general mechanism called a **data type** for defining new data structures that consist of many alternatives.

**Creating values:**

- `type my_bool = True | False`
- `type color = Blue | Yellow | Green | Red`

```ocaml
let b1 : my_bool = True
let b2 : my_bool = False
let c1 : color = Yellow
let c2 : color = Red
```

Use constructors to create values.
Using data type values:

```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 ->
```

use pattern matching to determine which color you have; act accordingly
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 -> print_string "blue"
    | Yellow -> print_string "yellow"
    | Green -> print_string "green"
    | Red -> print_string "red"
```
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 -> 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
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 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
```
• 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

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

Square s = 

Ellipse (r1, r2) = 

RtTriangle (s1, s2)  = 

RtTriangle [v1; ...;v5] = 

Type abbreviations can aid readability
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

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

Square builds a shape from a single side

RtTriangle builds a shape from a pair of sides
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 ->
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 -> ???
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

![Diagram of a convex polygon with vertices v1, v2, v3, v4, v5, and a triangle formed by v1, v2, v3 with the area of the polygon calculated as the sum of the areas of the triangle and the remaining 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
```
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) ->

Again, the type definition specifies the cases you must consider.
type key = int
type value = string

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)
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:

```plaintext
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 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
PARAMETERIZED TYPE DEFINITIONS
type ('key, 'val) tree =
  Leaf
| Node of 'key * 'val * ('key, 'val) tree * ('key, 'val) tree

type 'a stree = (string, 'a) tree

type sitree = int stree

**General form:**

**Definition:**

```plaintext
type 'x f = body
```

**Use:**

```plaintext
arg f
```

**A more conventional notation would have been (but is not ML):**

**Definition:**

```plaintext
type f x = body
```

**Use:**

```plaintext
f arg
```
Take-home Message

• Think of parameterized types like functions:
  – a function that take a type as an argument
  – produces a type as a result

• Theoretical basis:
  – System F-omega
  – a typed lambda calculus with general type-level functions as well as value-level functions
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.
Example Type Design

- 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**

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

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

type doc = elt list
```
Example Data

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

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

type doc = elt list

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 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”.

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

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

type doc = elt list
```
Challenge

- Change all of the "Arial" fonts in a document to "Courier".

```plaintext
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:

```plaintext
let rec chfonts (elts:doc) : doc =
```
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 ->
```
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)
```
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 =
```
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”.

```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) -> 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".

```
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`:

```
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:

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

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

- 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
```
Better Style

• Updated code:

```ocaml
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
```

Warning 8: this pattern-matching is not exhaustive. Here is an example of a value that is not matched:
TTFont _
Better Style

• Updated code, fixed:

```haskell
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