Lecture 13:

Polymorphism + Datatypes

Polymorphism:

Same code works
on
multiple types
of data

Type constructors:

a way of making types from smaller types

A int list is

[], or

x::xs where x:int
xs:int list

A string list is

[], or

x::xs where
x:string
xs:string list

An 'a list is

- [], or
- x::xs where x:'a
xs:'a list

bool list [true, false]

(int list) list [[1, 2], [3, 4]]

List<Bool>

```
fun length (l:int list):int =
  case l of
    [] => 0
  | x::xs => 1+length xs
```

```
fun length (l:string list):int
  case l of
    [] => 0
  | x::xs => 1+length xs
```

```
length [1,2]
```

'a = int

```
length ["a", "b"]
```

'a = string

for any type 'a ....

```
fun length( l: 'a list):int =
  case l of
    [] => 0
  | x::xs => 1+ length(xs)
```

fun sum(l: $\overset{'a}{\cancel{int}}$ list): int =

case l of

[] => 0

$| \overset{'a}{x} :: \overset{'a\ list}{xs} \Rightarrow \overset{'a}{x} + \underbrace{sum(xs)}_{int}$

type error: 'a + int ???

sum[true, false, true] ???

'a = bool

fun zip(l₁: ~~int~~ 'a list, l₂: ~~string~~ 'b list): (~~int~~ 'a * ~~string~~ 'b) list =

    case (l₁, l₂) of

      ([], _) => []

    | (_, []) => []

    | (x::xs, y::ys) => (x,y) :: zip(xs, ys)

zip([1,2], ["a", "b"])

'a = int   'b = string

for all 'a, 'b, 'c . . .

$$\text{fun append}\left(l_1 : \overset{\overset{\text{'a list}}{\text{'a}}}{\cancel{\text{int}}}\ \text{list},\ l_2 : \underline{\overset{\overset{\text{'a list}}{\text{'b}}}{\cancel{\text{int}}}\ \text{list}}\right) : \overset{\overset{\text{'a list}}{\text{'c}}}{\cancel{\text{int}}}\ \text{list} =$$

$$\text{case } l_1 \text{ of}$$

$$[\,] \Rightarrow \underline{l_2}$$

$$\mid \overset{\text{'a}}{x} :: \overset{\text{'a list}}{xs} \Rightarrow \underbrace{\underline{x} :: \text{append}(xs, l_2)}_{\text{'a list}}$$

'a = 'c

'b = 'c

Type inference

```
                 'a list    'a list     'a list
                 :'d        :'e       :'f
fun append( l1 ,     l2 ) =
  case l1 of
      []  =>  l2
       :'a   :'a list
    | x  ::  xs  =>  x :: append(xs, l2)
```

'd = 'a list

'e = 'f

'f = 'a list

Datatypes

An int list is

- [], or
- x::xs where x:int
  xs:int list

→ and that's it!

```
datatype int list =
    []
  | :: of int * int list

      ::(x,xs)
```

A tree is

Empty

Node(l,x,r)

where
l:tree
x:int
r:tree

→ and that's it!

```
datatype tree =
   Empty
 | Node of tree*int*tree
```

```
datatype bool =
    true
  | false
```

$(e_1, e_2) : T_1 * T_2$

where

$e_1 : T_1$

$e_2 : T_2$

Datatype constructor:

- create values $1 :: (2 :: [\,])$
- pattern matching

case e of

$p_1 \Rightarrow e_1$

$\mid p_2 \Rightarrow e_2$

$\vdots$

$\mid p_n \Rightarrow e_n$

Patterns p are

- made from
  - variables
  - constructors
- match values

1. First match
2. Exhaustive non-exhaustive warning
3. Non-redundant

Pattern	Matching Value	Binds
1	1	nothing
x	anything v	x to v
_	anything	nothing
(p_1, p_2)	(v_1, v_2) when p_1 matches v_1 p_2 matches v_2	what p_1 and p_2 bind
[]	[]	nothing
$p_1 :: p_2$	$v_1 :: v_2$	p_1 and p_2 bind
Empty	Empty	nothing
Node p	Node v	what p does

```
case 1::(2::[]) of
    [] => ____0____
  | [x] => ____1____
  | x::(y::ys) => ____2____
```

```
case ([],[]) of
    ([], _) => 1
  | (_, []) => 2
  | (x::xs, y::ys) => 3
```

↦ 1 first-match

or

~~↦ 2~~

```
case [1] of
    [] => 0
  | x::(y::ys) => 1
```

raises Match exception

non-exhaustive:

not all values match some pattern

```
case   of
    x => 1
  | y => 2
  | _ => 3
```

⟼ 1

redundant code

```
([], ) =>
([], x::xs) =>
```

error

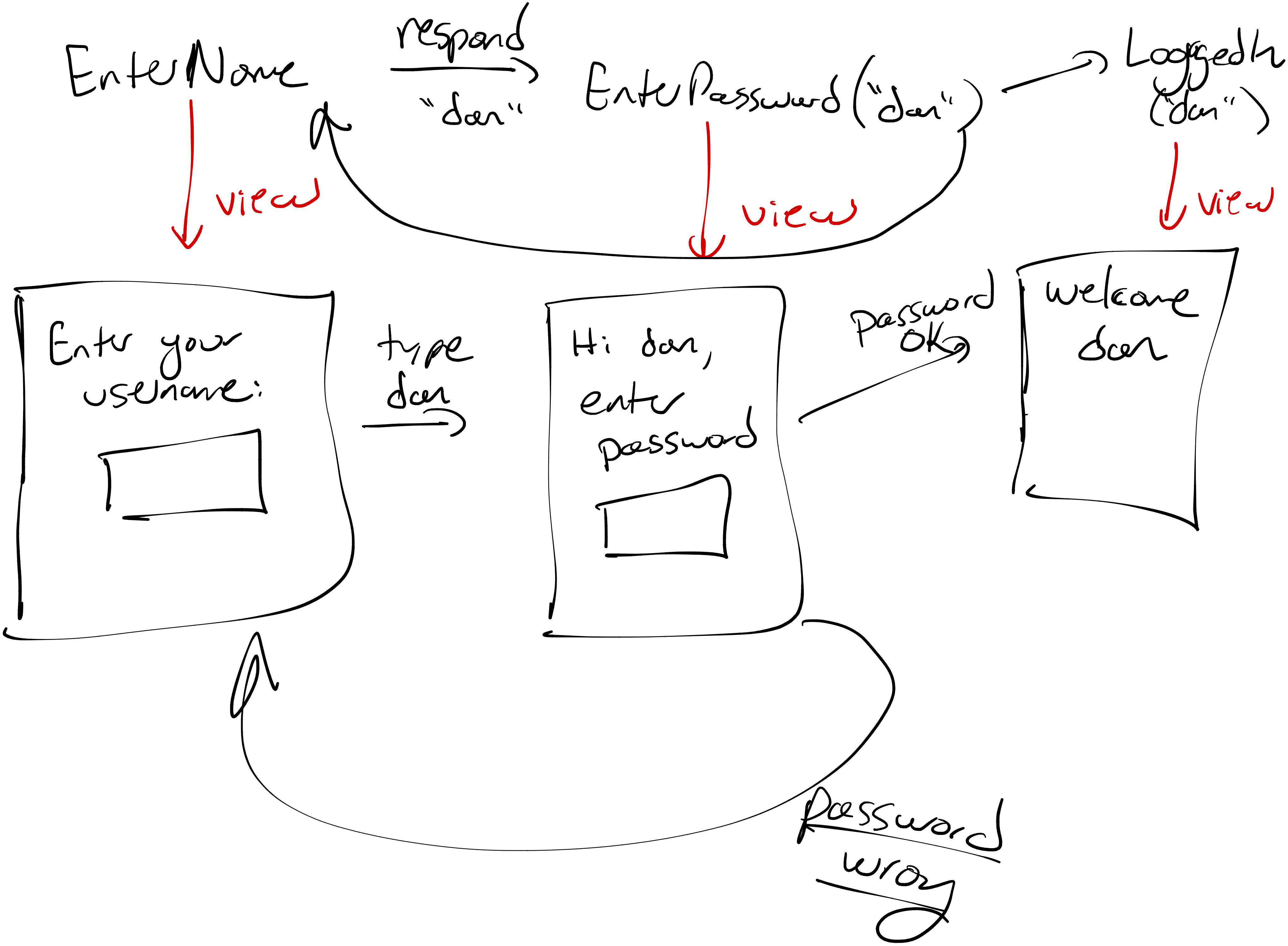
Interactive applications:

model - view - respond - controller

Separate the logic

from the display/

front-end (s)


EnterName
respond
"dan"
EnterPassword("dan")
LoggedIn("dan")
view
view
view
Enter your username:
type dan
Hi dan, enter password
password OK
welcome dan
Password wrong

```
datatype model =
    EnterName
  | EnterPassword of string (* name *)
  | LoggedIn of string (* name *)
```

1. *view* the model as a string

2. *respond* to user input by transitioning to a new model

```
(* input is what the user typed in *)
fun respond(m: model, Input: string): model =
    case m of

        EnterName => EnterPassword(input)

      | EnterPassword(name) =>
            (case checkPassword(name, input) of
                true => LoggedIn(name)
              | false => EnterName
            )
      | LoggedIn => - - -
```

```
fun view(m: model): string =
    case m of
        EnterName => "Please enter your name"
    | EnterPassword(name) =>
        "Hi " ^ name ^ "Please enter your password"
    | LoggedIn (name) =>
        "Welcome" ^ name ^ " _ _ _ _ "
```

Controller:

In a loop,

1. display the view of the model

2. recieve user input and compute the next model based on the respond

HW: shopping cart

- add thngs to cart
- pay
- . . . .