1. Data types to model interactive applications — HW6

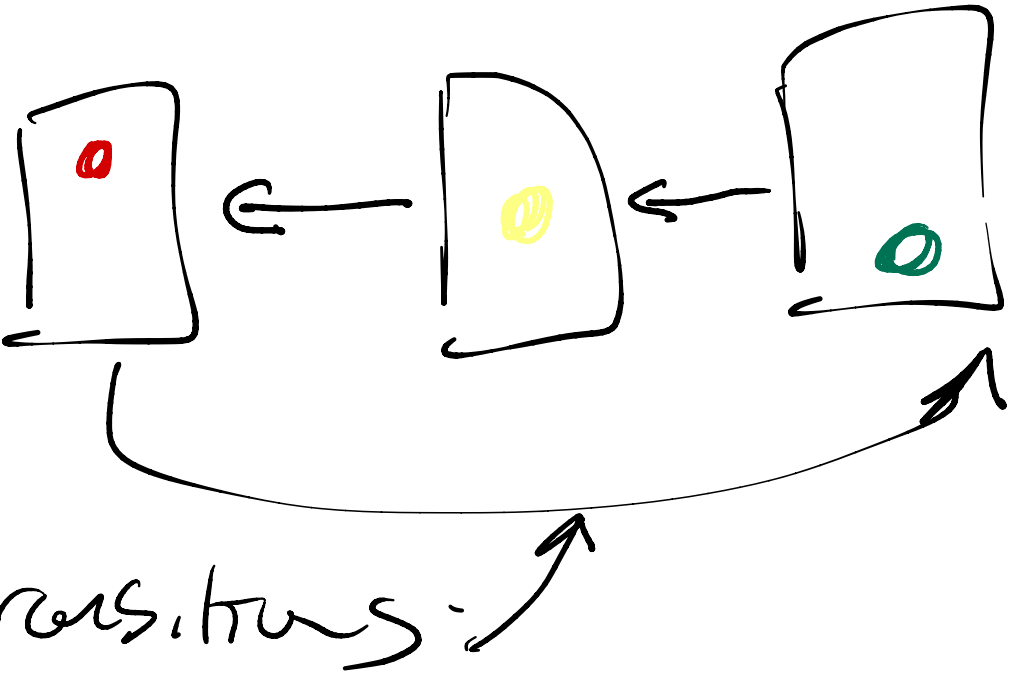
2. Polymorphism — HW7

Idea: model interactive time-varying application

as state-transition system

# Traffic light

States



time

transitions

# model - view - controller paradigm

separate

1. "data model"
2. view
3. responding to input/time/...

```
datatype model = Red
               | Green
               | Yellow
```

states

① view(s) ↪ web/iOS/android/native....

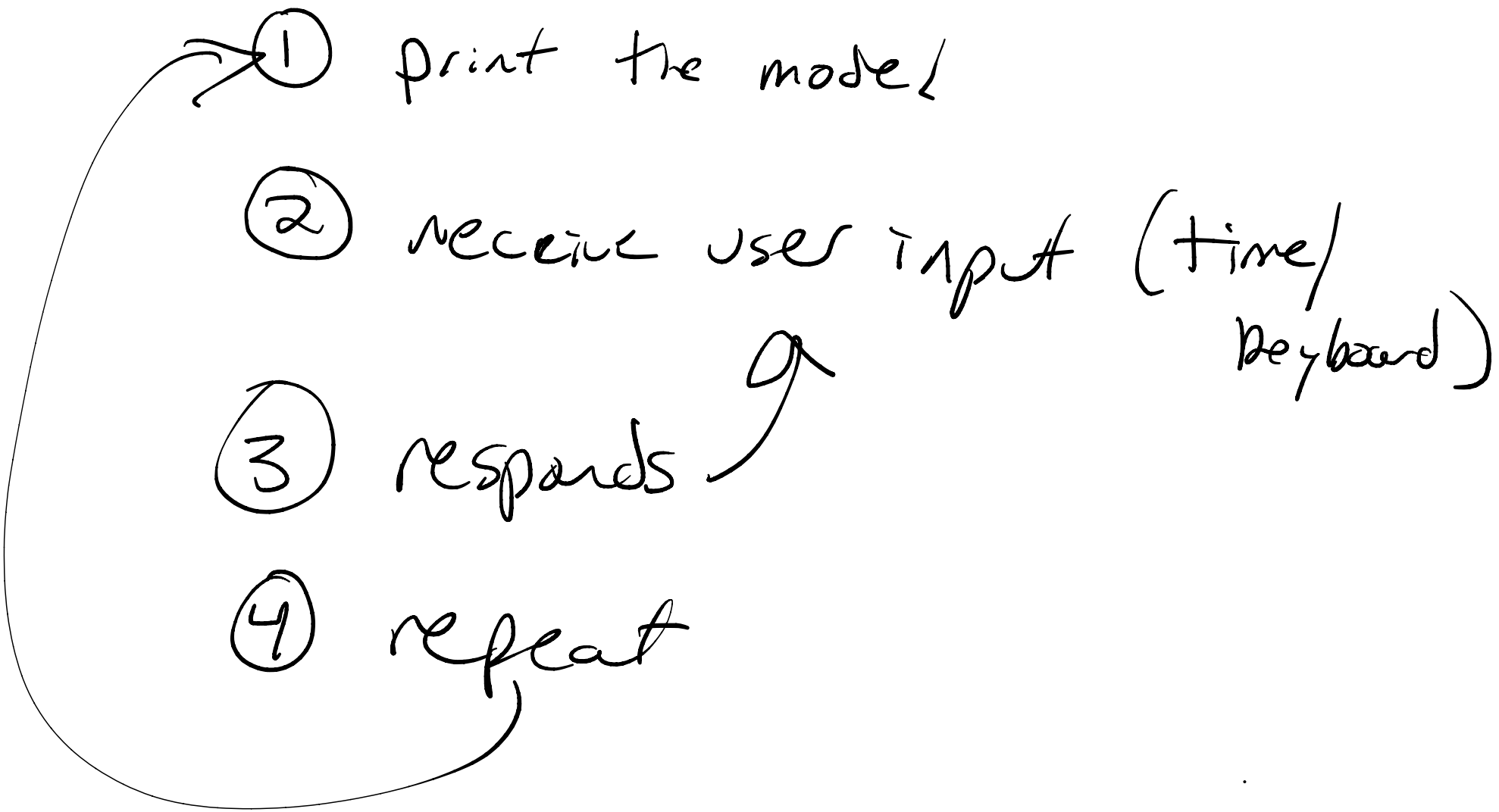
② transition model

```
fun view(m: model) : string =
    case m of
        Red => "
                 "
      | Yellow => "
                 "
      | Green => "
                 "
```

```
(* respond to time *)
fun respond(m: model): model =
    case m of
        Red    => Green
      | Yellow => Red
      | Green  => Yellow
```

says the "logic" in terms of the data model!

# Controller:


1 print the model
2 receive user input (time/keyboard)
3 responds
4 repeat

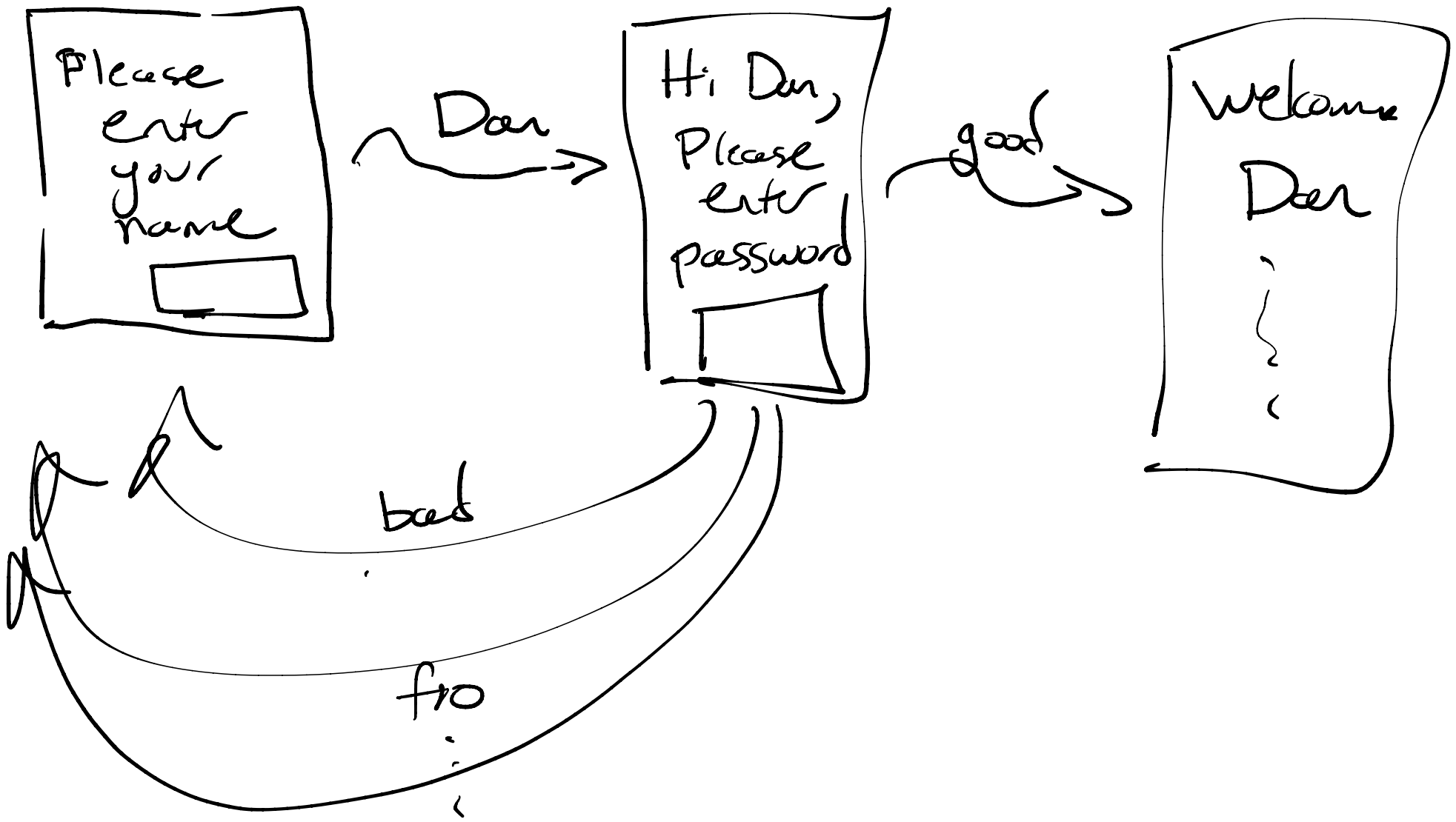
Controller( view, respond, Red );

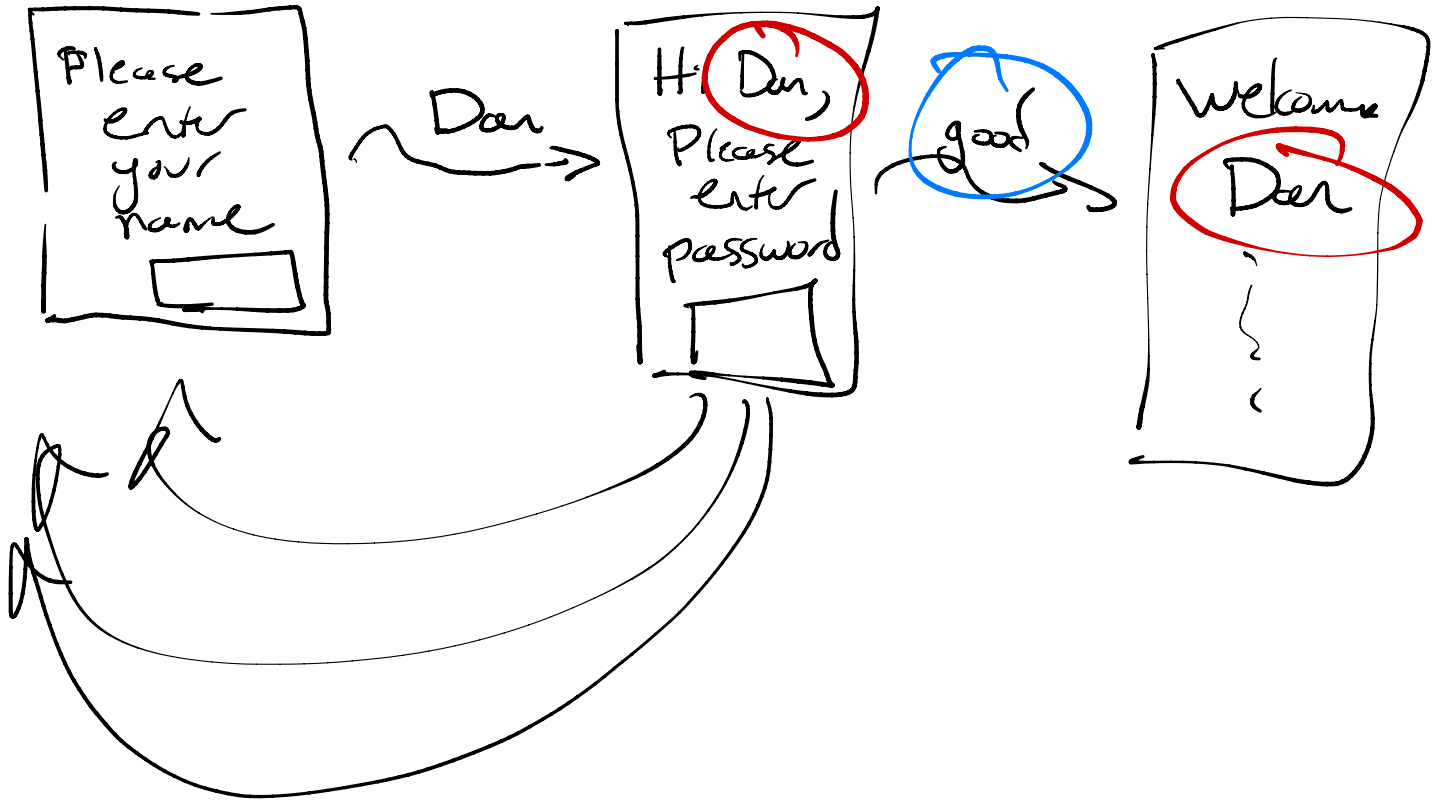
which state to start in

1. model datatype
2. view
3. respond
4. initial state
5. controller

specific to application

# Model with data: login prompt






name for State

EnterName

EnterPassword

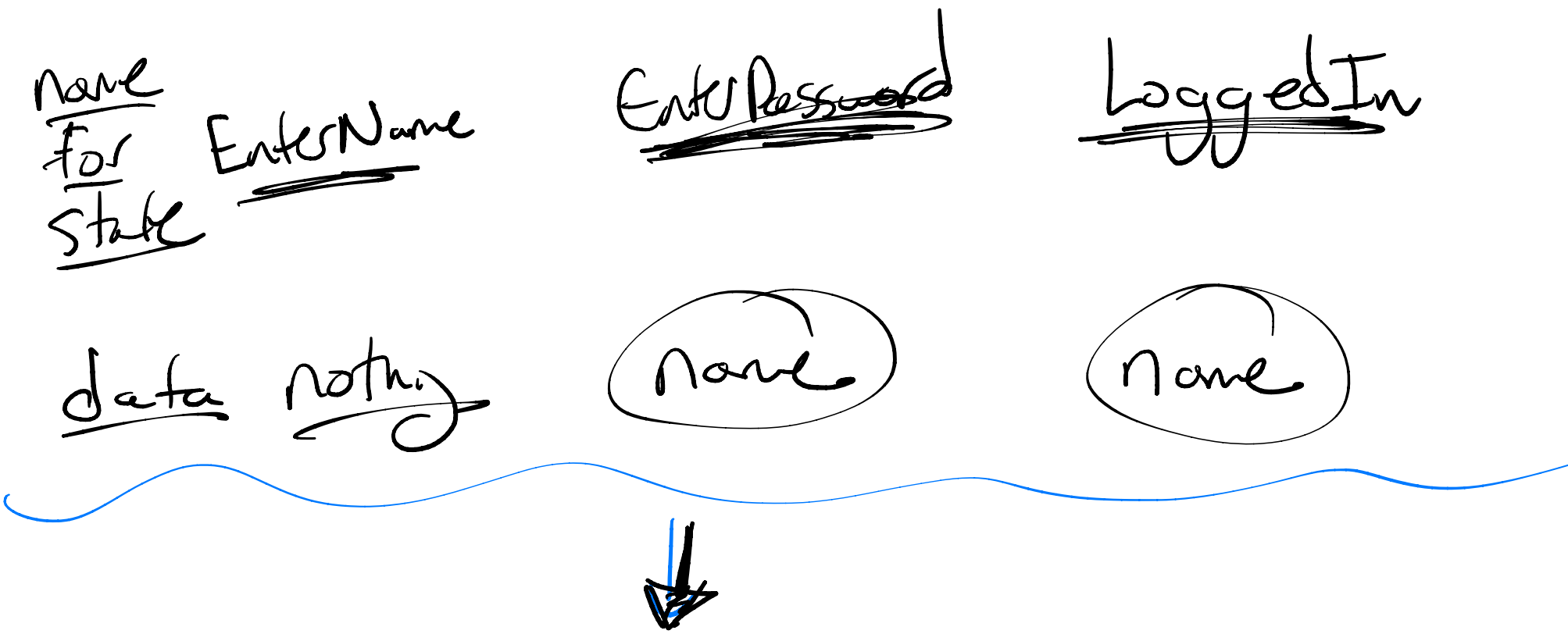
LoggedIn

data

nothing

none

none




```
datatype model = EnterName
               | EnterPassword of string
               | LoggedIn of string
```

```
datatype intlist = nil of 'int * int list
```

respond view initial State

EnterName

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

```
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(name) => ....
```

password

# HW6:

## Shopping cart

# Abstracting repeated code

Single point of control

# Polymorphism

→ Same code can work at many types

An int list is

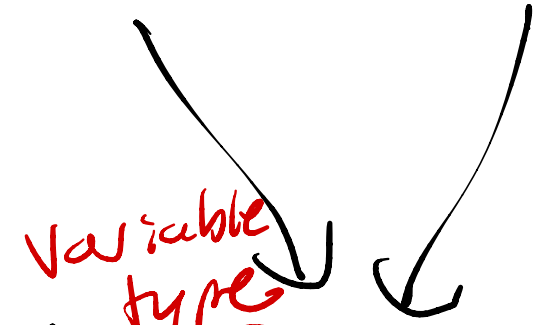
[] or

x::xs  x:int
xs:intlist

An string list is

[] or

x::xs  x:string
xs:string list




An 'a list is

[]

x::xs where x:'a
xs:'a list

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

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

```
length [1,2,3]
```

'a = int

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

'a = string

works for any type 'a

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

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

'a doesn't type check because specific int (real)

for any 'a  str int  for any 'b  str int  str int  str int

```
fun zip(l1 : 'a list, l2 : 'b list) : ('a * 'b) list =
  case (l1, l2) of
      ([], _) => []
    | (_, []) => []
    | (x::xs, y::ys) => (x,y) :: zip(xs, ys)
```

zip( [1,2,3], ["a","b","c"] )

'a = int
'b = string

zip( ["a"], ["d"] )
zip( [1,2,3], [4,5,6] )  ✓ ✓

for any

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

'a

'a

append( [1,2,3], ["a", "b"] ) = ???

'a = int

'b = string

# Type inference

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

'a = 'd list
'b = 'c
'c = 'd list

} Solve equations