lecture 07 – 2 main topics

- designing with lists of non-primitive types
- 2 types X and ListOfX
- reference and self-reference
- the reference rule and helpers

SPD Checklists

See full recipe page for details

```
(require spd/tags)
(require 2htdp/image)
(require 2htdp/universe)

;; My world program  (make this more specific)
(@htdw WS)
;; =================
;; Constants:


;; =================
;; Data definitions:

(@htdd WS)
;; WS is ... (give WS a better name)



;; =================
;; Functions:

(@htdf main)
(@signature WS -> WS)
;; start the world with (main ...)
;;
(@template-origin htdw-main)
(define (main ws)
  (big-bang ws          ;WS
    (on-tick   tock)    ;WS -> WS
    (to-draw   render) ;WS -> Image
    (on-mouse  ...)     ;WS Integer Integer MouseEvent -> WS
    (on-key    ...)))  ;WS KeyEvent -> WS

(@htdf tock)
(@signature WS -> WS)
;; produce the next ...
;; !!!
(define (tock ws) ws)

(@htdf render)
(@signature WS -> Image)
;; render ...
;; !!!
(define (render ws) empty-image)
```

HtDW

1. Domain analysis (use a piece of paper!)
    1. Sketch program scenarios
    2. Identify constant information
    3. Identify changing information
    4. Identify big-bang options
2. Build the actual program
    1. Constants (based on 1.2 above)
    2. Data definitions (based on 1.3 above)
    3. Functions
        1. main first (based on 1.4 and 2.2 above)
        2. wish list entries for big-bang handlers
    4. Work through wish list until done

```
on-tick
to-draw
on-key
on-mouse
```

HtDD

First identify form of information, then write:

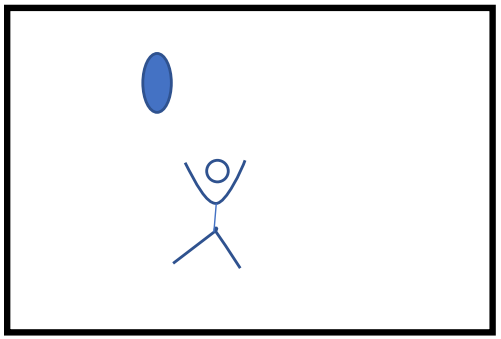
1. A possible structure definition (not until compound data)
2. A type comment that defines type name and describes how to form data
3. An interpretation to describe correspondence between information and data.
4. One or more examples of the data.
5. A template for a 1 argument function operating on data of this type.

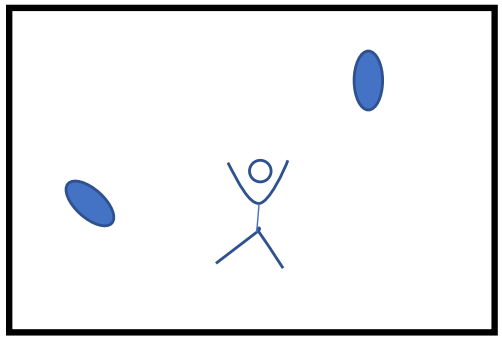
HtDF

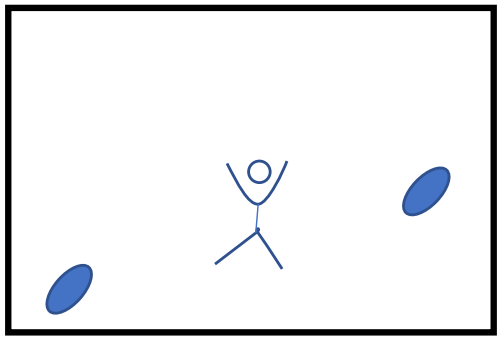
1. Signature, purpose and stub.
2. Define examples, wrap each in check-expect.
3. Template and inventory.
4. Code the function body.
5. Test and debug until correct

Test guidelines

1. at least 2
2. different argument/field values
3. code coverage
4. points of variation in behavior
5. 2 long / 2 deep







Constants:

width, height
egg image
y speed
r speed
background
egg image

Changing:

number of eggs

for each egg
  x
  y
  angle

big-bang options:

on-tick
to-draw
on-mouse

```
(@htdd Egg)
(define-struct egg (x y r))
;; Egg is (make-egg Number Number Number)
;; interp. the x, y position of an egg in screen coordinates (pixels),
;;         and rotation angle in degrees



(define (fn-for-egg e)
  (... (egg-x e)    ;Number
       (egg-y e)    ;Number
       (egg-r e)))  ;Number



(@htdd ListOfEgg)
;; ListOfEgg is one of:
;; - empty
;; - (cons Egg ListOfEgg)
;; interp. a list of eggs



(define (fn-for-loe loe)
  (cond [(empty? loe) (...)]
        [else
         (... (fn-for-egg (first loe))
              (fn-for-loe (rest loe)))]))
```

```
(define (next-eggs loe)
  (cond [(empty? loe) empty]
        [else
         ;temporarily ignore reference rule:
         ;(cons (make-egg (egg-x (first loe))
         ;                (+ (egg-y (first loe)) FALL-SPEED)
         ;                (+ (egg-r (first loe)) SPIN-SPEED))
         ;combination: cons is generic
         ;contribution: egg specific uses x, y and r
         ;---> combination in list fn, contribution in helper
         (cons (next-egg (first loe))
               (next-eggs (rest loe)))]))


(define (render-eggs loe)
  (cond [(empty? loe) MTS]
        [else
         ;temporarily ignore reference rule:
         ;(place-image (rotate (egg-r (first loe)) YOSHI-EGG)
         ;             (egg-x (first loe))
         ;             (egg-y (first loe))
         ;             (render-eggs (rest loe)))
         ;combination: place-image needs x, y
         ;contribution: uses r
         ;---> combination is in helper, also does contribution work
         (place-egg (first loe)
                    (render-eggs (rest loe)))]))
```