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

```
;; PART A
```

```
;PROBLEM 1: Evaluating expressions.  Note that in this question it is
 not necessary to show any intermediate steps - just provide the final
 value.

 a)  What is the value of the following expression?

 (+ (* (/ 6 2) (- 8 3)) 1)

 16


 b)  Given the partial function design:

 (define (foo b)
   (cond [(< b 0) (* b 2)]
         [(> b 10) (+ b 1)]
         [else b]))

 What is the value of the following expression?

 (foo 12)

 13


 c)  Given the following (non-sensical) partial data definition:

 (define-struct bar (a b c))
 ;; Bar is (make-bar (Integer Integer String))
 ;; interp. a bar with an a, b and c

 (define B1 (make-bar 1 2 "abc"))
 (define B2 (make-bar 3 4 "defg"))

 What is the value of the following expression?

 (+ (* (bar-b B1) (string-length (bar-c B1)))
    (* (bar-b B2) (string-length (bar-c B2))))


 22
```

```
; [3] marks
  - [1] mark each, no part marks
```

```
; PROBLEM 2:

 Design a function sign that consumes a number n and produces
 the sign of the number as a string.  So, the function produces
 "positive" if the number is bigger than 0, "zero" if the number
 is equal to zero and "negative" if the number is less than zero.

 In this question you must preserve the stub and template in your
 solution.  It's OK to put the check-expects after the stub, if
 you find it easier to structure your solution that way.

 Note: your function will produce a string.  It is not necessary
 to design a new data definition.
```

```
;; Number -> String
;; produces "positive", "zero" or "negative" to describe n
(check-expect (sign 15) "positive")
(check-expect (sign 0) "zero")
(check-expect (sign -10) "negative")

#;
(define (sign n) "zero")  ; stub

#;
(define (sign n)
  (... n))                ; template

(define (sign n)
  (cond [(> n 0) "positive"]
        [(= n 0) "zero"]
        [(< n 0) "negative"]))
```

```
[12] marks

 [2] correct signature
     - [1] for correct type consumed
     - [1] for correct type produced

 [2] accurate purpose statement

 [3] correct tests
     - [1] each for a test that produces
     "positive", "zero" and "negative"

 [1] stub
     - must produce value of correct type

 [1] template

 [3] body
     - [1] for each case
     OK to use nested if-expression
     (if (> n 0)
         "positive"
         (if (= n 0)
             "zero"
             "negative"))
```

```
Problem 3

 The following function design contains errors.  The function
 is supposed to consume a number and produce true if that
 number is greater than 100 and false otherwise.
```

```
;; Number -> Number
 ;; produces true if n is greater than 100, false otherwise
 (check-expect (greater-100? -10) false)
 (check-expect (greater-100? 100) false)
 (check-expect (greater-100? 10) true)

 (define (greater-100? n true))   ;stub

 (define (greater-100? n)
   (>= n 100))
```

```
Make the smallest number of changes possible so that the design of
 the function is consistent and correct.
```

```
;;; Number -> Boolean
 ;; produces true if n is greater than 100, false otherwise
 (check-expect (greater-100? -10) false)
 (check-expect (greater-100? 100) false)
 (check-expect (greater-100? 105) true)

 (define (greater-100? n) true)   ;stub

 (define (greater-100? n)
   (> n 100))
```

[4] marks
- [1] mark for changing signature to Number -> Boolean
- [1] mark for changing test value from 10 to some number greater than 100
- [1] mark for fixing parentheses in stub
- [1] mark for changing >= to > in body of function

```
;Problem 4:

 Given the following data definition:
```

```
(define-struct flight (bus econ))
;; Flight is (make-flight Integer Integer)
;; interp. a flight with bus seats available in business class
;; and econ seats available in economy class

(define F1 (make-flight 5 175))
(define F2 (make-flight 3 58))

(define (fn-for-flight f)
  (... (flight-bus f)
       (flight-econ f)))

;; Template rules used:
;; - compound: 2 fields
```

```
;
 Design a data definition for ListOfFlight
```

```
;; ListOfFlight is one of:
;; - empty
```

```
;; - (cons Flight ListOfFlight)
;; interp. a list of flights

(define LOF1 empty)
(define LOF2 (cons F1 (cons F2 empty)))

#;
(define (fn-for-lof lof)
  (cond [(empty? lof) (...)]
        [else
         (... (fn-for-flight (first lof))
              (fn-for-lof (rest lof)))]))

;; Template rules used:
;; - one of: 2 cases
;; - atomic distinct: empty
;; - compound: (cons Flight ListOfFlight)
;; - reference rule: (first lof) is Flight
;; - self-reference rule: (rest lof) is ListOfFlight
```

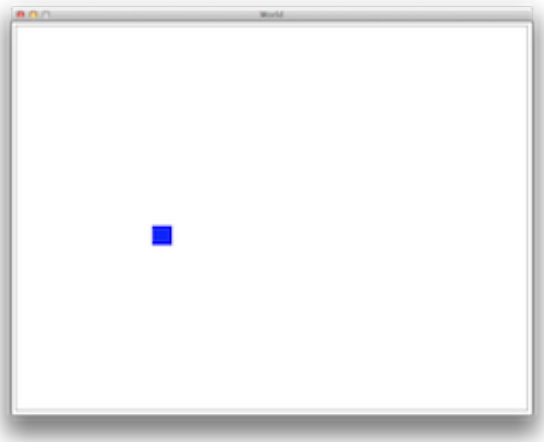
```
;[14] marks
 - [3] types comment
    (deduct [1] for each distinct error)
 - [1] interp
 - [2] at least two examples, one empty
 - [7] template and rules:
    - [2] 2 cases, cond w/ two [q a] clauses
    - [1] atomic distinct: empty
    - [1] compound rule
    - [2] reference rule & call to helper
    - [2] self-reference rule & natural recursion
```

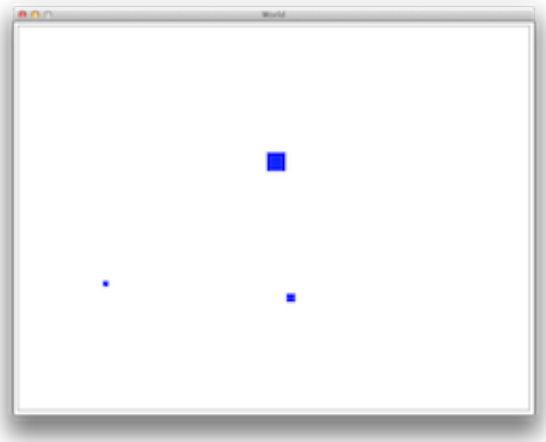
The remainder of this midterm involves the design of a complete world program. The desired behaviour of the world program is as follows:

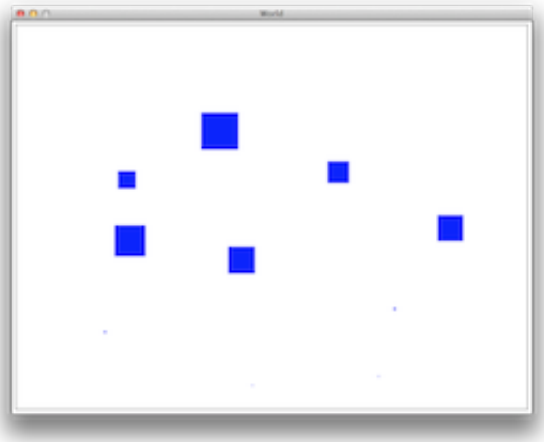
- there is a rectangular screen with fixed size
- there are an arbitrary number of blocks (squares) of the same colour
- a block is created wherever the mouse is clicked
- the initial size of each block is the same
- a block falls vertically at a constant rate
- the size of each block decreases to 90% of its previous value on each tick of the world
- note that blocks keep shrinking in size
- note that blocks keep falling even after they disappear off the bottom of the window - they do not stop when they reach the bottom of the window

PROBLEM 5: World program domain analysis

Conduct a complete domain analysis for this world program. To make it easier we have provided three images of the program taken at different times for you to use. Please identify constant information, changing information and required big-bang-options.







CONSTANT	CHANGING INFORMATION	big-bang OPTIONS
width	number of squares	to-draw
height	x coord of square	on-tick
initial size	y coord of square	on-mouse
background	size of square	
shrink rate		
colour of square		

```
;[4] - constants, one for each correct to maximum of 4
     - must not also be listed as changing information
 [4] - changing information, one for each correct
     - must not also be listed as constant
 [3] - big-bang options, one for each correct

 DEDUCT:
 [1] - for each extra changing information
 [1] - for each extra big-bang option
```

```
;
 PART 2 -
```

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

;; shrinking, falling blocks program.

;; =========================
;; Constants

(define WIDTH 800)
(define HEIGHT 600)

(define MAX-SIZE 100)      ;side-length of block when it is created
(define SHRINK-RATE .9)    ;size of each block is 90% of previous value on each tick
(define SPEED 2)           ;vertical speed of a block

(define BLOCK-COLOR "blue")
(define MTS (empty-scene WIDTH HEIGHT))
```

```
;; ========================
;; Data Definitions

(define-struct block (l x y))
;; Block is (make-block Number(0, MAX-SIZE] Integer Integer))
;; interp. a square block of side-length l at position (x, y)
(define B1 (make-block 50 0 10))
(define B2 (make-block 20.5 100 150))

#;
(define (fn-for-block b)
  (... (block-l b)
       (block-x b)
       (block-y b)))

;; Template rules used:
;; compound: 3 fields

;; ListOfBlock is one of:
;; - empty
;; - (cons Block ListOfBlock)
;; interp. a list of blocks
(define LOB1 empty)
(define LOB2 (cons (make-block 40 0 10)
                   (cons (make-block 50.1 100 150) empty)))

#;
(define (fn-for-lob lob)
  (cond [(empty? lob) (...)]
        [else
         (... (fn-for-block (first lob))
              (fn-for-lob (rest lob)))]))

;; Template rules used:
;; - one of: 2 cases
;; - atomic distinct: empty
;; - compound: (cons Block ListOfBlock)
;; - reference: (first lob) is Block
;; - self-reference: (rest lob) is ListOfBlock
```

R2

SR1

2

1

PROBLEM 6: On the two data definitions above you should do 4 things:

- draw appropriate reference and/or self-reference arrows
- label each arrow with R or SR
- also number each arrow with a number like 1, 2, 3 etc.
- on the template functions draw appropriate arrows that correspond to reference and/or self-reference arrows and label them with the number of the corresponding reference or self-reference arrow

```
;[7] marks

 [1] - correct self-reference arrow from ListOfBlock to ListOfBlock
 [1] - correct reference arrow from Block to Block
 [1] - both arrows correctly labelled
 [1] - both arrows numbered
 [1] - correct arrow from natural recursion to function definition
 [1] - correct arrow from helper function call to helper function
       definition
 [1] - both arrows on templates correctly labelled with numbers

 [1] mark deducted for each extra arrow / label
 [1] mark deducted if any arrow in wrong direction
```

```
;; =========================
;; Functions

;; ListOfBlock -> ListOfBlock
;; runs the shrinking blocks world; start with (main empty)
;; <no tests for main function>
(define (main lob)
  (big-bang lob
            (on-tick tick-lob)
            (to-draw render-lob)
            (on-mouse handle-mouse)))


;; ListOfBlock -> ListOfBlock
;; decrease size of blocks and move them
;; !!!
;(define (tick-lob lob) empty)


;; ListOfBlock -> Image
;; render all the blocks onto MTS
;; !!!
;(define (render-lob lob) MTS)


;; ListOfBlock Integer Integer MouseEvent -> ListOfBlock
;; creates a new block wherever the mouse is clicked
;; !!!
;(define (handle-mouse lob x y me) lob)
```

```
;PROBLEM 7: Function Design

Note that the design of handle-mouse has been completed for you.

Complete the design of the function tick-lob on the wish-list above.  Please
copy the signature and purpose then complete the design of the tick-lob function
as well as the design of any other function required by the design recipe.

DO NOT attempt to complete the design of render-lob - the other function on the
wish-list above.
```

```
;; ListOfBlock -> ListOfBlock
;; decrease size of all blocks
(check-expect (tick-lob empty) empty)
(check-expect (tick-lob (cons (make-block 10 25 35)
                              (cons (make-block 15 135 235) empty)))
              (cons (make-block (* 10 SHRINK-RATE) 25 (+ 35 SPEED))
                    (cons (make-block (* 15 SHRINK-RATE) 135 (+ 235 SPEED)) empty)))
#;
(define (tick-lob lob) empty)   ;stub

;; <template copied from ListOfBlock>

(define (tick-lob lob)
  (cond [(empty? lob) empty]
        [else
         (cons (tick-block (first lob))
               (tick-lob (rest lob)))]))

;; Block -> Block
;; produces new block with size SHRINK-RATE of given block
(check-expect (tick-block (make-block 10 25 35))
              (make-block (* 10 SHRINK-RATE) 25 (+ 35 SPEED)))

#;
(define (tick-block b) b)

;; <template copied from Block>

(define (tick-block b)
  (make-block (* (block-l b) SHRINK-RATE)
              (block-x b)
              (+ (block-y b) SPEED)))
```

```
;[8] - Q7a
 ;; tick-lob
 [1] check-expect on empty list -> must be first test
 [1] check-expect on list of size at least 2
 [1] SHRINK-RATE & SPEED used in check-expects
 [1] base-case result correct
 [1] cons used at combination step
 [1] natural recursive call intact
 [1] helper function called on (first lod)
 [1] helper function appropriately named

 [11] - Q7b
 ;; tick-block
 [2] signature
    - [1] type consumed is correct
    - [1] type produced is correct
 [1] clear purpose statement
 [1] correct check-expect with SHRINK-RATE and SPEED used
 [1] (define (tick-block b)
 [1] function clearly based on template for Block
     (contains (block-l b), (block-x b), (block-y b))
 [2] make-block
   - deduct [1] if incorrect # or type of arguments passed
 [1] (* (block-l b) SHRINK-RATE)
 [1] (block-x b)
 [1] (+ (block-y b) SPEED)
```

```
;;====================================
```

```
;Functions below needed to run but did not have to be written during
 exam
```

```
;; ListOfBlock -> Image
;; renders list of blocks as image
(check-expect (render-lob empty) MTS)
(check-expect (render-lob (cons (make-block 10 25 35)
                                (cons (make-block 15 135 235) empty)))
              (place-image (square 10 "solid" BLOCK-COLOR)
                           25 35
                           (place-image (square 15 "solid" BLOCK-COLOR)
                                        135 235
                                        MTS)))
; <template copied from ListOfBlock>
(define (render-lob lob)
```

```
  (cond [(empty? lob) MTS]
        [else
         (place-block (first lob) (render-lob (rest lob)))]))


;; Block Image -> Image
;; places block b onto image i to produce a new image
(check-expect (place-block (make-block 10 25 35) MTS)
              (place-image (square 10 "solid" BLOCK-COLOR) 25 35 MTS))

;(define (place-block b i) MTS)

;; <template copied from Block>

(define (place-block b i)
  (place-image (square (block-l b) "solid" BLOCK-COLOR)
               (block-x b)
               (block-y b)
               i))

;; ListOfBlock Integer Integer MouseEvent -> ListOfBlock
;; creates a new block wherever the mouse is clicked
(check-expect (handle-mouse empty 10 30 "drag") empty)
(check-expect (handle-mouse empty 10 30 "button-down")
              (cons (make-block MAX-SIZE 10 30) empty))

;(define (handle-mouse lob x y me) lob)   ;stub

(define (handle-mouse lob x y me)
  (cond [(string=? me "button-down")
         (cons (make-block MAX-SIZE x y) lob)]
        [else lob]))
```