Assignment 1
Data definitions
Intervals
Use an *interval* when the information to be represented is numbers within a certain range.

Integer[0, 10] is all the integers from 0 to 10, inclusive.
Number[0, 10) is all the numbers from 0 inclusive to 10 exclusive.
[ and ] include the end point
( and ) do not include the end point
Problem

Imagine that you’re designing a program to manage ticket sales for a theater, which happens to be perfectly rectangular.

Design a data definition to represent a seat number in a row, where each row has 32 seats. (Just represent the seat number, not the row number.)
SeatNum is Natural[1, 32]
interp. seat numbers in a row; 1 and 32 are aisle seats

(define SN1 1)
(define SN2 12)
(define SN3 32)

Template for SeatNum functions:
(define (fn-for-seat-num sn)
  (... sn))
Design a function that produces true if the given seat number is on the aisle.
;; Data definitions:

;; SeatNum is Natural[1, 32]
;; interp. seat numbers in a row; 1 and 32 are aisle seats
(define SN1 1)
(define SN2 12)
(define SN3 32)
;; (define (fn-for-seat-num sn)
;;   (... sn))

;; Functions:

;; aisle? : SeatNum -> Boolean
;; Produce true if the given seat number is on the aisle.
(define aisle? sn) #false)

(check-expect (aisle? SN1) #true)
(check-expect (aisle? SN2) #false)
(check-expect (aisle? SN3) #true)
;; Data definitions:

;; SeatNum is Natural[1, 32]
;; interp. seat numbers in a row; 1 and 32 are aisle seats
(define SN1 1)
(define SN2 12)
(define SN3 32)

;; (define (fn-for-seat-num sn)
;;   (... sn))

;; Functions:

;; aisle? : SeatNum -> Boolean
;; Produce true if the given seat number is on the aisle.
(define aisle? sn)
  (or (= sn 1)
      (or (= sn 1)
          (= sn 32)))

(check-expect (aisle? SN1) #true)
(check-expect (aisle? SN2) #false)
(check-expect (aisle? SN3) #true)
Enumerations
Use an *enumeration* when the information to be represented consists of a fixed number of distinct items.
Problem

As part of designing a system to keep track of student grades, you are asked to *design a data definition* to represent the letter grade in a course, which is one of A, B, or C.
LetterGrade is one of:
- "A"
- "B"
- "C"
interp. the letter grade in a course

<examples are redundant for enumerations>

Template for LetterGrade functions:
(define (fn-for-letter-grade lg)
  (cond [(string=? lg "A") (...)]
        [(string=? lg "B") (...)]
        [(string=? lg "C") (...)]))
Problem

*Design a function* that consumes a letter grade and produces the next highest letter grade.
;; Data definitions:

;; LetterGrade is one of:
;; – "A"
;; – "B"
;; – "C"
;; interp. the letter grade in a course

;; (define (fn-for-letter-grade lg)
;;   (cond [(string=? lg "A") (...)]
;;         [(string=? lg "B") (...)]
;;         [(string=? lg "C") (...)]))

;; Functions:

;; bump-up : LetterGrade → LetterGrade
;; Produce next highest letter grade (no change for A)
(define (bump-up lg) "A")

(check-expect (bump-up "A") "A")
(check-expect (bump-up "B") "A")
(check-expect (bump-up "C") "B")
;; Data definitions:

;; [...]

;; (define (fn-for-letter-grade lg)
;;   (cond [(string=? lg "A") (...)]
;;         [(string=? lg "B") (...)]
;;         [(string=? lg "C") (...)]))

;; Functions:

;; bump-up : LetterGrade -> LetterGrade
;; Produce next highest letter grade (no change for A)
(define (bump-up lg)
  (cond [(string=? lg "A") "A"
        [(string=? lg "B") "A"
         [(string=? lg "C") "B"]]))

(check-expect (bump-up "A") "A")
(check-expect (bump-up "B") "A")
(check-expect (bump-up "C") "B")
Itemizations
Use an *itemization* when domain information is comprised of two or more subclasses, *at least one of which is not a distinct data item*. 
Problem

Consider designing the system for controlling a New Year’s Eve countdown display.

*Design a data definition* to represent the current state of the countdown, which falls into one of three categories:

- not yet started
- from 10 to 1 seconds before midnight
- complete (i.e., Happy New Year! 🎉)
(define CD1 #false)
(define CD2 10) ; just started
(define CD3 1) ; almost over
(define CD4 "complete")

;; (define (fn-for-countdown c)
;;     (cond [(false? c) (...)]
;;           [(number? c) (... c)]
;;           [else (...)]))
In this course, when you write (in the signature) the type of data a function takes, you can assume that this is what it will be given.

Racket doesn’t enforce this – it’s possible for you to pass in bad data by mistakes – but other programming languages will.

This is why we could just write else as our last case rather than check that it’s the string "complete" – and why we didn’t check that the number is between 1 and 10.
Problem

*Design a function* that consumes Countdown and produces an image showing the current status of the countdown.
(require 2htdp/image)

;; Data definitions:

;; CountDown is one of:
;; - #false
;; - Natural[1, 10]
;; - "complete"
;; [...]

;; Functions:

;; countdown->image : Countdown -> Image
;; Produce an image of the current state of the countdown
(define (countdown->image c) (square 0 "solid" "white"))

(check-expect ...)
There should be at least as many tests as there are cases in the itemization.

If there are intervals in the itemization, then there should tests at all points of variance in the interval. In the case of adjoining intervals, it’s critical to test the boundaries.
(require 2htdp/image)

;; Data definitions:

;; CountDown is one of:
;; - #false
;; - Natural[1, 10]
;; - "complete"
;; [...]

;; Functions:

;; countdown->image : Countdown -> Image
;; Produce an image of the current state of the countdown
(define (countdown->image c) (square 0 "solid" "white"))

(check-expect (countdown->image #false) (square 0 "solid" "white"))
(check-expect (countdown->image 5) (text (number->string 5) 24 "black"))
(check-expect (countdown->image "complete") (text "Happy New Year!" 24 "red"))
(require 2htdp/image)

;; Data definitions: [...]

;; Functions:

;; countdown-to-image : Countdown -> Image
;; Produce an image of the current state of the countdown
(define (countdown->image c)
  (cond [(false? c) (...)]
        [(number? c) (... c)]
        [else (...)]))

(check-expect (countdown->image #false)
              (square 0 "solid" "white"))
(check-expect (countdown->image 5)
              (text (number->string 5) 24 "black"))
(check-expect (countdown->image "complete")
              (text "Happy New Year!" 24 "red"))
(require 2htdp/image)

;; Data definitions: [...]  
;; Functions:

;; countdown-to-image : Countdown -> Image
;; Produce an image of the current state of the countdown
(define (countdown->image c)
  (cond
   [(false? c) (square 0 "solid" "white")]
   [(number? c) (text (number->string c) 24 "black")]
   [else (text "Happy New Year!" 24 "red")]))

(check-expect (countdown->image #false) (square 0 "solid" "white"))
(check-expect (countdown->image 5) (text (number->string 5) 24 "black"))
(check-expect (countdown->image "complete") (text "Happy New Year!" 24 "red"))
Don’t be fooled by these examples – in real programs, it would be very strange to have a new data definition for each function!

Usually there are many functions that consume the type defined by a data definition.
Worlds
DrRacket provides a mechanism for supporting interactive programs called **big-bang**.
big-bang: Less Sheldon Cooper, more creation-of-the-universe
[big-bang.rkt]
Think again about a countdown program. What needs to happen?

Imagine we generate one tick per second:

<table>
<thead>
<tr>
<th>Tick #</th>
<th>n</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>Happy New Year!</td>
</tr>
</tbody>
</table>
Each time the clock ticks, call `next-countdown` with the current world state to get the next world state.

Each time the clock ticks, call `countdown->image` with the current world state to draw the current world state.

Type of argument or signature of function

; CountDown
; CountDown -> CountDown
; CountDown -> Image

Initial world state

(big-bang 10
  (on-tick next-countdown)
  (to-draw countdown->image))
**big-bang** puts all the pieces together to start the world in motion.
(require 2htdp/image)
(require 2htdp/universe)

;; Constants
(define EMPTY (empty-scene ...))

;; Data definitions:

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

;; Functions:

;; main : WS -> WS
;; Start the world with ...
(define (main ws)
  (big-bang ws
    (on-tick tock) ; WS -> WS
    (to-draw render) ; WS -> Image
    (stop-when ...) ; WS -> Boolean
    (on-mouse ...) ; WS Integer Integer MouseEvent -> WS
    (on-key ...))) ; WS KeyEvent -> WS

;; WS -> WS
;; Produce the next ...
;; !!!
(define (tock ws) 0)

;; WS -> Image
;; Render ...
;; !!!
(define (render ws) EMPTY)
Templates aren’t just for beginners!

After this course, you won’t always template very simple functions, but the idea of templates – getting the basic structure of a piece of code before you get to the details – is something to keep using!
Problem

Design a world program that represents a countdown. The program should display the number of seconds remaining and should decrease at each clock tick. On reaching zero, it should stay there and not change.
Problem

Design a world program that represents a countdown. The program should display the number of seconds remaining and should decrease at each clock tick. On reaching zero, it should stay there and not change.

[See countdown.rkt]
Acknowledgments

This lecture incorporates material from:

- Matthias Felleisen
- Robert Bruce Findler
- Matthew Flatt
- Gregor Kiczales
- Shriram Krishnamurthi