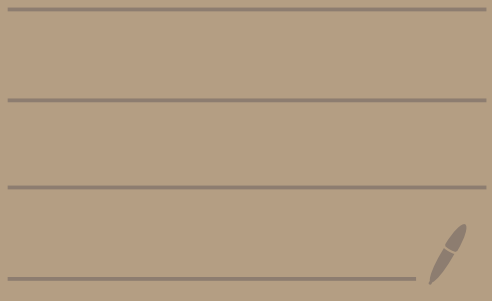
Racket: How to Code - Simple Data (UBC)



The complete How to Code course consists of 2 parts, each of which is 6 weeks long. Each week consists of 1 or 2 modules, and those modules all have a similar structure, comprised of:

- An overview describing the module learning goals and summarizing the work required to complete the module.
- A number of blended topic lectures, consisting of video interspersed with questions for you to answer.
- A set of problems that will let you practice the new design techniques before the quiz.
- Several discussion questions. These questions help organize the general discussion forums by creating a section for each module. The discussion forums are a great resource to see what other students had to say about the material you are working on. The course staff will also be active in the disussion forums.
- A module quiz. The module quiz is a self-assessed design problem.
- A module wrap up.

The following chart provides an overview of the course topics:

Week	Module Name	Lectures	Time to complete	Practice Problems	Quiz
	Overall learning goal				
1	**Beginning Student Language**	8	5-8 Hours	4	*Self-Assessed Design Quiz*
	Learn to program with the core of the programming language used throughout the course.				
	How to Design Functions (HtDF) Recipe	6	4-7 Hours	3	
	Learn to use the HtDF recipe to design functions that consume simple primitive data.				
2	**How to Design Data (HtDD) Recipe**	12	5-8 Hours	3	*Self-Assessed Design Quiz*
	Learn to use the HtDD recipe to design data definitions for atomic data.				
3	**How to Design Worlds (HtDW) Recipe**	7	3-5 Hours	1	*Self-Assessed Design Quiz*
	Compound Data	3	4-6 Hours	3	
	Learn to use the HtDW recipe to design interactive programs with atomic and then compound world state.				
4	**Self-Reference**	7	5-7 Hours	4	*Multiple Choice Quiz*
	Learn how to use well-formed self-referential data definitions to represent arbitrary sized data.				
	Reference	3	4-6 Hours	2	
	Learn to predict and identify the correspondence between references in a data definition and helper function calls in functions that operate on the data.				
5	**Naturals**	2	3-4 Hours	2	*Self-Assessed Design Quiz*
	Design an alternate data definition for the natural numbers, and learn to write functions using this new data definition.				
	Helpers	6	6-9 Hours	1	
	Learn a set of rules for designing functions with helper functions.				
6	**Binary Search Trees**	6	5-6 Hours	3	*Multiple Choice Quiz*
	Design a data definition for Binary Search Trees, and learn to write functions operating on BSTs.				

Lecture 1 – Racket Beginning Student Language (BSL)

Part A – Expressions & Evaluation

① Install "DrRacket" (functional programming IDE). "Racket" is an offshoot of Scheme, born from Lisp. We use "Beginning Student" language in this lecture.

② Basic Expression Syntax for Racket: "> (<primitive> <expression> ...)" "> <value>"

※ space here

e.g. > (+ 3 4) e.g. > (sqrt 16)

③ Comment: 3.1 put ";" in front e.g. > ;(+ 3 4)

3.2 menu → "Insert" → "Insert Comment Box" and write lines + pictures + ... in the box

④ example of finding $\sqrt{3^2+4^2}$: > (sqrt (+ (* 3 3) (* 4 4))) (left shortens spaces to the extreme)

can omit the space due to "("

must have spaces because "*3"/"*4" makes NO sense

> (sqrt (+ (sqr 3) (sqr 4))) (same as above)

square (3^2)

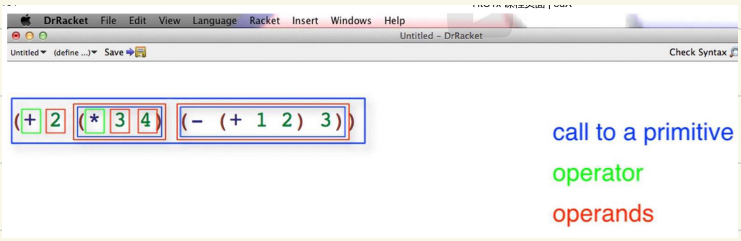
⑤ meaning of "#i": indicates an irrational number can NOT be written as $\frac{a}{b}$, where a,b are integers; that means its decimal representation is infinitely long and NOT repeating.

e.g. > (sqrt 2) outputs "#i1.4142135623730951" (inexact number shown)

⑥ be careful about order of values in expressions

e.g. > (/ (+ 4 6.2 -12) 3) stands for $(4+6.2+(-12)) \div 3 = -0.6$

⑦ Evaluation Rules that Racket uses to evaluate BSL expressions




```
(+ 2 (* 3 4) (- (+ 1 2) 3))
(+ 2 12      (- (+ 1 2) 3))
(+ 2 12      (- 3       3))
(+ 2 12      0)
14
```

Intuitively:
left to right
inside to outside

```
Welcome to DrRacket, version 5.3.
Language: Intermediate Student; m
14
>
```

To evaluate a primitive call:
- first reduce operands to values
- then apply primitive to the values

⑧ ";" will comment afterwards contents in the current line.

"#;" will comment the nearest next function (whole function)

Part B - Strings & Images

① **String concatenation**: "string-append" primitive call

e.g. > (string-append "Ada" " " "Lovelace") output: "Ada Lovelace"

② Racket doesn't support diff. types' concatenation ⇒ only string type can be the operands.

③ **String slicing**: "substring" primitive call indices same rule as other languages

e.g. > (substring "0123456" 2 4) output: "23" (include starting index, exclude last index)

④ **Making Images starter**: > (require 2htdp/image) tells DrRacket we want to use the image functions from the 2nd edition of the How to Design Programs book

⑤ example codes of making images: > (require 2htdp/image) (must starter)

> (circle 10 "solid" "red") > (rectangle 30 60 "outline" "blue")

> (text "hello" 24 "orange") > (above/beside/overlay (circle 10 "solid" "red")
(circle 20 "solid" "yellow")
(circle 30 "solid" "green"))

Part C - Constants

① Define constant variable syntax

```
(require 2htdp/image)

(define WIDTH 400)
(define HEIGHT 600)

(* WIDTH HEIGHT)

(* 400 HEIGHT)

(* 400 600)

240000
```

Welcome to DrRacket, version 5.3.1 [3m].
Language: Beginning Student; memory limit: 128 MB.
>

To form a constant definition:

(define <name> <expression>)

open parentheses — close parentheses

sequence of characters including:
a..z A..Z 0..9
! @ $ % ^ & * _ + - = ? < >

for example:
hello? HI! the-dog cat<=?

Evaluation rules for constant definitions

- to evaluate a constant definition:
  evaluate the expression and record the resulting value as the value of the constant with the given name
- to evaluate a defined constant name:
  the value is the recorded value

② can define Constant variable as an image

Untitled (define ...)

```
(require 2htdp/image)

(define WIDTH 400)
(define HEIGHT 600)

(define CAT )

(define RCAT (rotate -10 CAT))
(define LCAT (rotate  10 CAT))
```

Part D - Functions

① Creation syntax

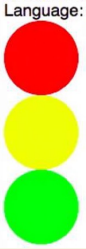
> (define (<fName> <varName>)
  <expression>)

```
(require 2htdp/image)

;; function-definitions-starter.rkt


;(above (circle 40 "solid" "red")
;       (circle 40 "solid" "yellow")
;       (circle 40 "solid" "green"))

(define (bulb c)
  (circle 40 "solid" c))

(above (bulb "red") (bulb "yellow") (bulb "green"))
```

function name

Using a function makes this code more concise; if the function is named well, it gives the code more meaning as well.

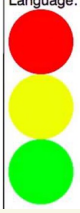
```
Welcome to DrRacket, version 5.3.1 [3m].
Language: Beginning Student; memory limit: 128 MB.
```



② Functions related evaluation

```
(require 2htdp/image)

;; function-definitions-starter.rkt


(define (bulb c)
  (circle 40 "solid" c))

(bulb (string-append "re" "d"))
(bulb "red")
(circle 40 "solid" "red")
```

```
Welcome to DrRacket, version 5.3.1 [3m].
Language: Beginning Student; memory limit: 128 MB.
```



⇨ To evaluate primitive call
- first reduce operands to values (called the arguments)
- then apply primitive to the values

For function definitions
- simply record definition

To evaluate function call
- first reduce operands to values (called the arguments)
- replace function call by
  - body of function in which every occurrence of parameter(s) are replaced by corresponding argument

Part E - Booleans and if Expressions

① true ; false (all lowercases) are values of Boolean type variables.

② relational operators include "=" ">" "<" ">=" "<="

③ examples: > (string=? "foo" "bar") output: false determine whether strings "foo" & "bar" are equal for all chars.

```
> (require 2htpd/image)
  (define I1 (rectangle 10 20 "solid" "red")
        I2 (rectangle 20 10 "solid" "blue"))
  (< (image-width I1) (image-width I2))
```

output: true

④ if expressions syntax:

To form an if expression:

```
(if <expression>   ← question (must produce boolean)
    <expression>   ← true answer
    <expression>)  ← false answer
```

⑤ if expressions evaluation

```
(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 20 10 "solid" "blue"))

(if (< (image-width I2)
       (image-height I2))
    (image-width I2)
    (image-height I2))

(if (< (image-width ▬)
       (image-height I2))
    (image-width I2)
    (image-height I2))

(if (< 20
       (image-height I2))
    (image-width I2)
    (image-height I2))

(if (< 20
       (image-height ▬))
    (image-width I2)
    (image-height I2))

(if (< 20 10)
    (image-width I2)
    (image-height I2))

(if false
    (image-width I2)
    (image-height I2))

(image-height I2)
```

To evaluate an if expression:

- if the question expression is not a value evaluate it, and replace with value
- if the question is true replace entire if expression with true answer expression
- if the question is false replace entire if expression with false answer expression
- the question is a value other than true or false so produce an error

⑥ "and" "or" "not" primitive calls ⇒ boolean outputs

>1 arguments 1 argument only

Part F - Stepper in DrRacket

① Show procedural runs step by step in a dynamic and graphical way.

(≈ visualized debugger)

Part G - Discovering Primitives

① Right click on "primitive calls" → "Search in Help Desk for "..." " → Read the Documentation

② "guess" primitive calls

③ Racket BSL is not a strongly typed language (unlike Java).

Lecture 2 - How to design Functions (HtDF)

Part A - Full Speed HtDF Receipe

① HtDF
- Signature, purpose, stub
- Examples (wrapped in check-expect)
- Inventory - template & constants
- Code body
- Test and debug

```
PROBLEM:

Design a function that consumes a number and produces twice that number.
Call your function double. Follow the HtDF recipe and show the stub and
template.

;; Number -> Number                          Signature
;; produce 2 times the given number          Purpose
(check-expect (double 3) 6)                  Examples/tests
(check-expect (double 4.2) (* 2 4.2))

;(define (double n) 0)  ;this is the stub     Stub

;(define (double n)     ;this is the template Template
;  (... n))

(define (double n)                           Function body
  (* 2 n))
```

How to Design Functions (HtDF)
To design a single function.
- Signature, purpose, stub
- Examples (wrapped in check-expect)
- Inventory - template & constants
- Code body
- Test and debug

```
Welcome to DrRacket, version 5.3 [3m].
Language: Intermediate Student with lambda; memory limit: 512 MB.
Both tests passed!
```

1: yell-starter.rkt | 2: area-starter.rkt | 3: image-area-start... | 4: tall-starter.rkt

```
(require 2htdp/image)
;; Image -> Boolean
;; produce true if the image is tall (height is greater than width)
(check-expect (tall? (rectangle 2 3 "solid" "red")) true)
(check-expect (tall? (rectangle 3 2 "solid" "red")) false)
(check-expect (tall? (rectangle 3 3 "solid" "red")) false)

;(define (tall? img) false)    ;stub

;(define (tall? img)           ;template
;  (... img))

;(define (tall? img)
;  (if (> (image-height img) (image-width img))
;      true
;      false))

(define (tall? img)
  (> (image-height img) (image-width img)))
```

```
Welcome to DrRacket, version 5.3.1 [3m].
Language: Beginning Student; memory limit: 128 MB.
All 3 tests passed!
```

(if ANYTHING true false) — ANYTHING

always equivalent expressions

true & false can be replaced by #t & #f

Lecture 3 - How to design Data

Part A - cond Expressions

① "cond" is a multi-armed conditional. It can have any number of cases all at same level.

```
#;
(define (aspect-ratio img)
  (if (> (image-height img) (image-width img))
      "tall"
      (if (= (image-height img) (image-width img))
          "square"
          "wide")))

(define (aspect-ratio img)
  (cond [Q A]
        [Q A]
        [Q A]))
```

( ) balance each other
[ ] balance each other

Both are equivalent, but by convention we use [ ] around question/answer pairs in cond. This makes the cond easier to read.

To form cond expression:

(cond [<expression> <expression>] ...)

question, answer

one or more question answer pairs

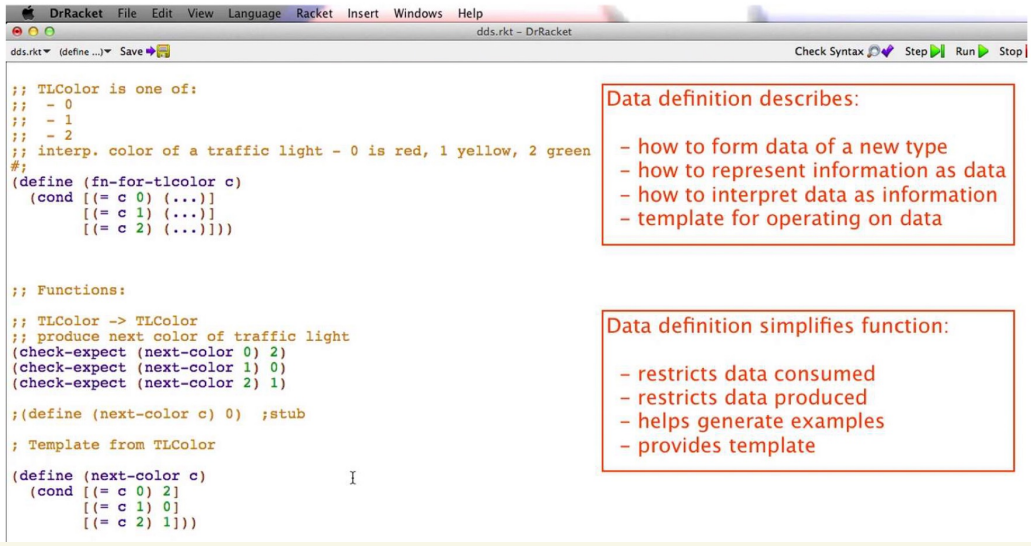
Each question must evaluate to a boolean.
Last question can be else.

② an example:

```
(define (aspect-ratio img)
  (cond [(> (image-height img) (image-width img)) "tall"]
        [(= (image-height img) (image-width img)) "square"]
        [else "wide"]))
```

Part B - Data Definitions

① **Data Definitions**



Part C - How to design data recipe (HtDD)

① Atomic Non-Distinct Data example with Data Driven Templates Rules

```
;; CityName is String
;; interp. the name of a city
(define CN1 "Boston")
(define CN2 "Vancouver")
#;
(define (fn-for-city-name cn)
  (... cn))

;; Template rules used:              For the first part of the course
;;   atomic non-distinct: String     we want you to list the template
;;                                   rules used after each template.
;;

;; Functions:

;; CityName -> Boolean
;; produce true if the given city is Hogsmeade
(check-expect (best? "Boston") false)
(check-expect (best? "Hogsmeade") true)

;(define (best? cn) false)  ;stub

; took template from CityName
#;
(define (best? cn)
  (if (string=? cn "Hogsmeade")
      true
      false))


(define (best? cn)
  (string=? cn "Hogsmeade"))
```

? needed for boolean outcome functions

(if <expression> true false)

can be simplified to

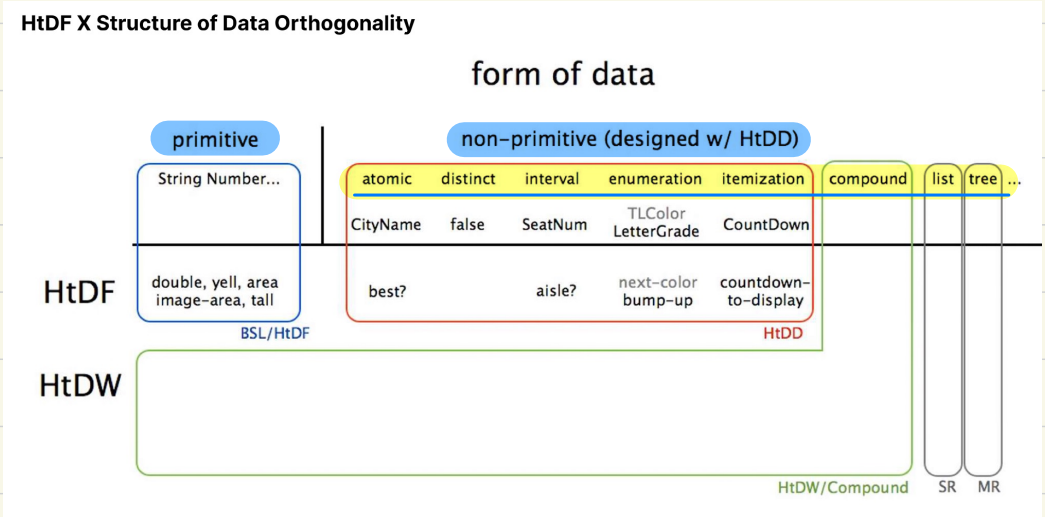
<expression>

② HtDF X Structure of Data Orthogonality

the same HtDF (and HtDW) recipes work with all forms of data

the recipe is mostly **orthogonal*** to the form of data

so you get the cross-product easily



Part D - Interval, Enumeration & Itemization

① ex of interval: build data for Natural [1, 32]

② ex of enumeration: build data for LetterGrade, which is one of: "A"; "B"; "C"

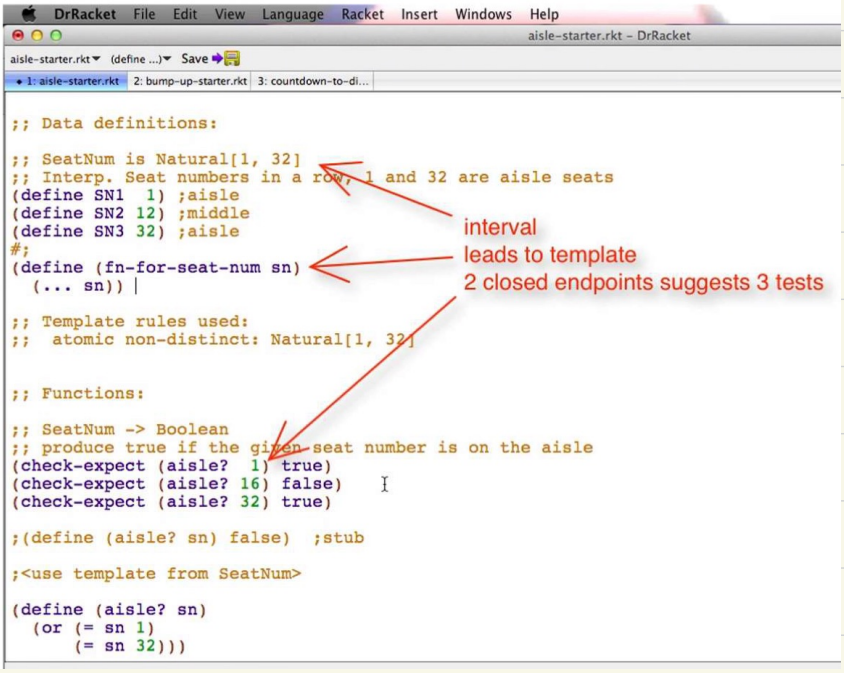
③ itemization: use an itemization when domain information is comprised of 2 or more subclasses, at least one of which is not a distinct data item.

ex: build data for CountDown, which is one of: false / Natural [1, 10] / "complete"

(mixed data → guards needed: (number? c))

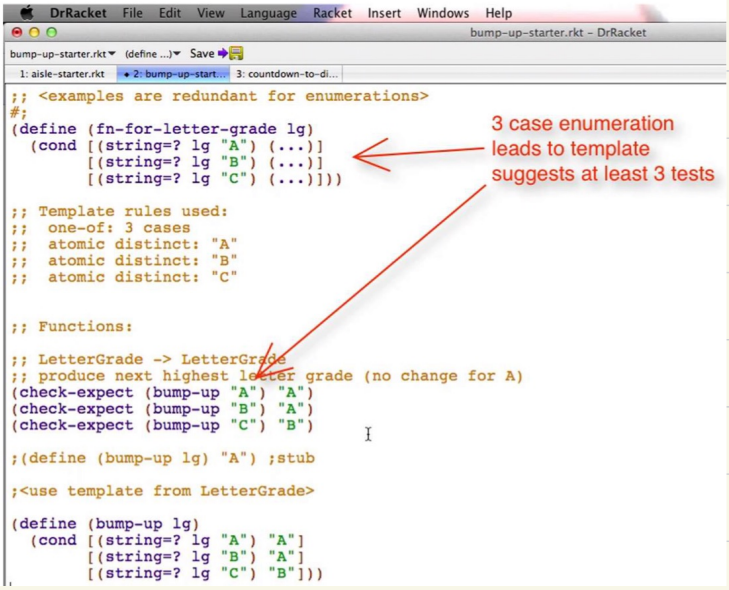
④ HtDF with Interval

HtDF with Interval



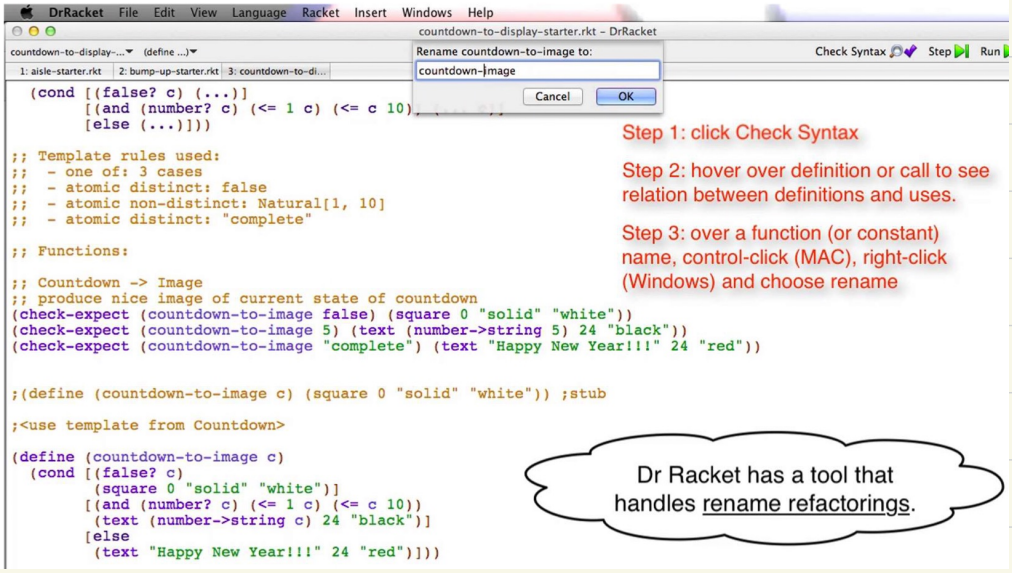
⑤ HtDF with Enumeration

HtDF with Enumeration



⑥ HtDF with Itemization

HtDF with Itemization



Lecture 4 – How to Design Worlds (interactive program)

Part A – Big-bang Mechanism

① Moving cat example - Idea

(x-axis, left to right)

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

;; Cat is Number
;; interp. x coordinate of cat
;;
(define C1 0)
(define C2 (/ WIDTH 2))
#;
(define (fn-for-cat c)
  (... c))
;; Template rules used:
;;  - atomic non-distinct: Number


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

;; Cat -> Cat
;; increase cat x position by SPEED
(check-expect (next-cat 0) SPEED)
(check-expect (next-cat 100) (+ 100 SPEED))
#;
(define (next-cat c) 1)           ; stub

;; <use template from Cat>

(define (next-cat c)
  (+ c SPEED))


;; Cat -> Image
;; add CAT-IMG to MTS at proper x coordinate and CTR-Y
(check-expect (render-cat 100)
              (place-image CAT-IMG 100 CTR-Y MTS))
#;
(define (render-cat c) MTS)  ; stub

;; <use template from Cat>

(define (render-cat c)
  (place-image CAT-IMG c CTR-Y MTS))
```

place-image puts CAT-IMG at the x,y coordinate c and CTR-Y on MTS

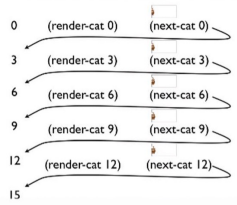
tick #	x	image
0	0	
1	3	
2	6	
3	9	
...		
312	936	
313	939	
314	942	

28 ticks per second

A function that puts the image of the cat at the right place on the empty scene (MTS).

1-1 "big-bang" expression

big-bang is polymorphic:
it works for any type of world state



expression that evaluates to initial world state

```
(big-bang 0                    ; X
          (on-tick next-cat)   ; X -> X
          (to-draw render-cat)); X -> Image
```

for any given use of big-bang, all the X have to be the same type

each time the clock ticks, call next-cat with the current world state to get the next world state

each time the clock ticks, call render-cat with the current world state to draw the current world state

Part B – How to Design Worlds Receipe (HTDW)

① Domain Analysis

1-1 Sketch program scenarios

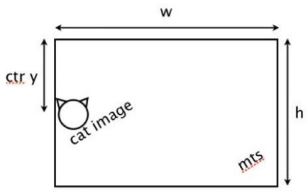
1-2 Identify constant information

1-3 Identify changing information

1-4 Identify big-bang options

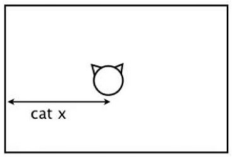
①' Moving cat example (simple)

Note: In computer science, 0 coordinates are set at the left-top corner of the system.



Constant

width
height
ctr-y
mts (background)
cat image



Changing

x coordinate of cat

If your program needs to:	Then it needs this option:
change as time goes by (nearly all do)	on-tick
display something (nearly all do)	to-draw
change in response to key presses	on-key
change in response to mouse activity	on-mouse
stop automatically	stop-when

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 2.1 Constants 2.2 Data definitions

2.3 Functions < main function first
wish list entries for big-bang handlers

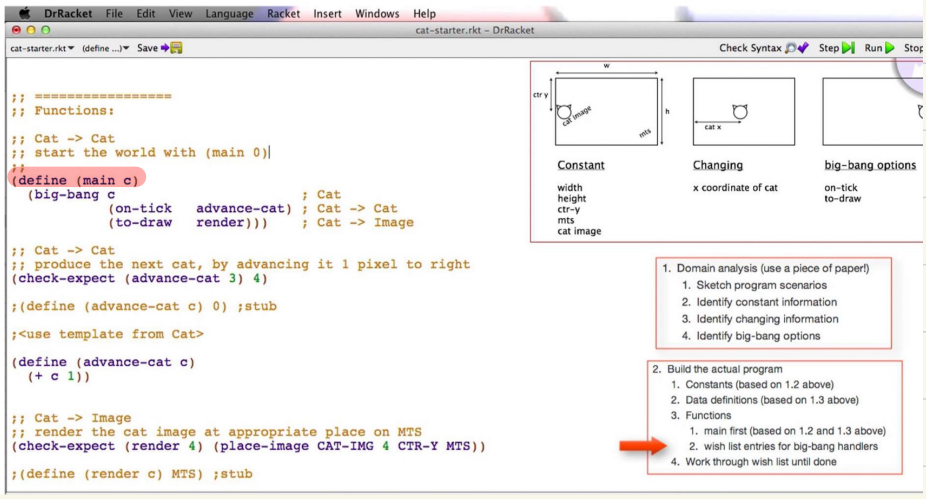
2.4 work through wish list until done.

②' Moving Cat example

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

;; A cat that walks from left to right across the screen.
;; ==============
;; Constants:

(define WIDTH 600)
(define HEIGHT 400)
(define CTR-Y (/ HEIGHT 2))
(define MTS (empty-scene WIDTH HEIGHT))
```

```
(define CAT-IMG )

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

;; Cat is Number
;; interp. x position of the cat in screen
(define C1 0)             ;left edge
(define C2 (/ WIDTH 2))   ;middle
(define C3 WIDTH)         ;right edge
#;
(define (fn-for-cat c)
   (... c))
;;Template rules used:
;;  - atomic non-distinct: Number
```

Working through the Wish List




```
;<use template from Cat>

(define (render c)
  (place-image CAT-IMG c CTR-Y MTS))
```

we can run it by
> (main 0)

Part C - Improving a World Program

① Idea: Programs are always changing. (so elements s.a. constants variables help a lot)

② Moving cat example < add SPEED > (define SPEED 3)
add key handler

```
(define (main c)
  (big-bang c                        ; Cat
            (on-tick   advance-cat)  ; Cat -> Cat
            (to-draw   render)       ; Cat -> Image
            (on-key    handle-key))) ; Cat KeyEvent -> Cat
```

```
(check-expect (handle-key 10 " ") 0)
(check-expect (handle-key 10 "a") 10)
(check-expect (handle-key  0 " ") 0)
(check-expect (handle-key  0 "a") 0)

;(define (handle-key c ke) 0) ;stub

(define (handle-key c ke)  ;two parameters c & ke
  (cond [(key=? ke " ") 0]
        [else c]))
```

Lecture 5 - Compound Data

Part A - define-struct

① Syntax : > (define-struct <name> (<name>...))

```
(define-struct pos (x y))

(define P1 (make-pos 3 6))
(define P2 (make-pos 2 8))

(pos-x P1) ;3
(pos-y P2) ;8

(pos? P1)       ;true
(pos? "hello") ;false
```

constructor

selectors

predicate

A structure definition defines:

constructor: make-<structure-name>
selector(s): <structure-name>-<field-name>
predicate: <structure-name>?

(define-struct pos (x y)) defines:

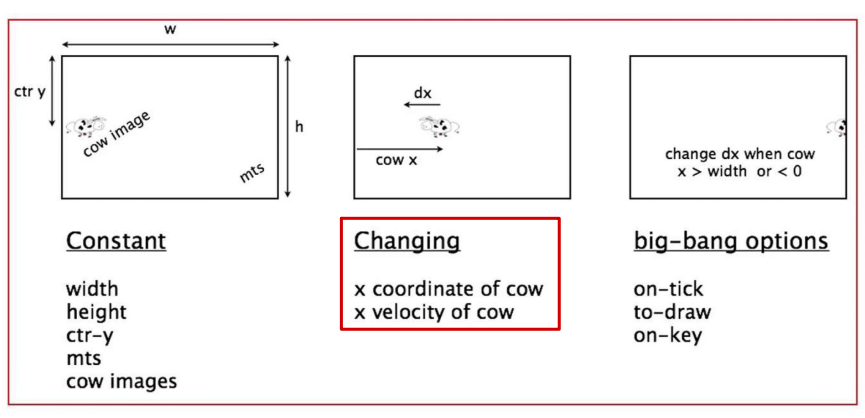
constructor: make-pos
selectors: pos-x pos-y
predicate: pos?

Part B - "Worlds" with Compound Data (Cow example)

① Compound Data definition : a data consists of two or more items that naturally belong together

② "Cow" example domain

2 changing variables need compound data structure



Codes in next 2 pages

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

;;   cowabunga-starter.rkt  problem statement
;;   cowabunga-v0.rkt       has constants
;;   cowabunga-v1.rkt       has data definition
;;   cowabunga-v2.rkt       has main function, wish list entries
;; > cowabunga-v3.rkt       has next-cow
;;   cowabunga-v4.rkt       has render-cow
;;   cowabunga-v5.rkt       has handle-key

;; A cow, meandering back and forth across the screen.


;; =================
;; Constants:

(define WIDTH  400)
(define HEIGHT 200)


(define CTR-Y (/ HEIGHT 2))


(define RCOW  )
(define LCOW  )


(define MTS (empty-scene WIDTH HEIGHT))




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

(define-struct cow (x dx))
;; Cow is (make-cow Natural[0, WIDTH] Integer)
;; interp. (make-cow x dx) is a cow with x coordinate x and x velocity dx
;;         the x is the center of the cow
;;         x  is in screen coordinates (pixels)
;;         dx is in pixels per tick
;;
(define C1 (make-cow 10  3)) ; at 10, moving left -> right
(define C2 (make-cow 20 -4)) ; at 20, moving left <- right
#;
(define (fn-for-cow c)
  (... (cow-x c)    ;Natural[0, WIDTH]
       (cow-dx c))) ;Integer

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




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

;; Cow -> Cow
;; called to make the cow go for a walk; start with (main (make-cow 0 3))
```

```
;; no tests for main function
(define (main c)
  (big-bang c
            (on-tick next-cow)       ; Cow -> Cow
            (to-draw render-cow)     ; Cow -> Image
            (on-key  handle-key)))   ; Cow KeyEvent -> Cow



;; Cow -> Cow
;; increase cow x by dx; when gets to edge, change dir and move off by 1
(check-expect (next-cow (make-cow 20           3)) (make-cow (+ 20 3)  3)) ;away
from edges
(check-expect (next-cow (make-cow 20          -3)) (make-cow (- 20 3) -3))

(check-expect (next-cow (make-cow (- WIDTH 3)  3)) (make-cow WIDTH     3))
;reaches edge
(check-expect (next-cow (make-cow 3           -3)) (make-cow 0        -3))

(check-expect (next-cow (make-cow (- WIDTH 2)  3)) (make-cow WIDTH    -3)) ;tries
to pass edge
(check-expect (next-cow (make-cow 2           -3)) (make-cow 0         3))

;(define (next-cow c) c)      ;stub

(define (next-cow c)
  (cond [(> (+ (cow-x c) (cow-dx c)) WIDTH) (make-cow WIDTH (- (cow-dx c)))]
        [(< (+ (cow-x c) (cow-dx c)) 0)     (make-cow 0     (- (cow-dx c)))]
        [else
         (make-cow (+ (cow-x c) (cow-dx c))
                   (cow-dx c))]))


;; Cow -> Image
;; place appropriate cow image on MTS at (cow-x c) and CTR-Y
(check-expect (render-cow (make-cow 99 3))
              (place-image RCOW 99 CTR-Y MTS))
(check-expect (render-cow (make-cow 33 -3))
              (place-image LCOW 33 CTR-Y MTS))
;(define (render-cow c) MTS)  ;stub

; took template from Cow
(define (render-cow c)
  (place-image (choose-image c) (cow-x c) CTR-Y MTS))

;; Cow -> Image
;; produce RCOW or LCOW depending on direction cow is going
(check-expect (choose-image (make-cow 10  3)) RCOW)
(check-expect (choose-image (make-cow 11 -3)) LCOW)
(check-expect (choose-image (make-cow 11  0)) LCOW)
;;
(define (choose-image c)
  (if (> (cow-dx c) 0)
      RCOW
      LCOW))

;; Cow KeyEvent-> Cow
;; reverse direction of cow travel when space bar is pressed
;; !!!
(check-expect (handle-key (make-cow 10 1) " ") (make-cow 0 1))
(check-expect (handle-key (make-cow 10 1) "a") (make-cow 10 1))
(check-expect (handle-key (make-cow  0 3) " ") (make-cow 0 3))
(check-expect (handle-key (make-cow  0 3) "a") (make-cow 0 3))
;(define (handle-key c ke) c) ;stub
(define (handle-key c ke)
  (cond [(key=? ke " ") (make-cow 0 (cow-dx c))]
        [else c]))
```

Lecture 5 – Self-Reference

Part A – Lists (arbitrary sized data)

① "cons" expression defines a list (can mix different types) : > (cons <ele1> <ele2> ...)

first|non-first element(s) of a list "L1" : > (first L1) > (rest L1)

whether empty or not : > (empty? empty) (true) > (empty? L1) (false)

```
(define L1 (cons "Flames" empty))                ; a list of 1 element
(define L2 (cons 10 (cons 9 (cons 10 empty))))   ; a list of 3 elements
(define L3 (cons (square 10 "solid" "blue")
                 (cons (triangle 20 "solid" "green")
                       empty)))

(first L1)
(first L2)
(first L3)

(rest L1)
(rest L2)
(rest L3)

(first (rest L2))          ;how do I get the second element of L2
(first (rest (rest L2)))   ;how do I get the third element of L2

(empty? empty)
```

cons a two argument constructor
first selects the first element of a list
rest selects the elements after the first
empty? produce true if argument is the empty list

```
Language: Beginning Student; memory limit: 128 MB.
empty
"Flames"
10
■
empty
(cons 9 (cons 10 empty))
(cons ▲ empty)
9
10
true
```

② Self-reference

```
;; ListOfString is one of:
;;  - empty
;;  - (cons String ListOfString)
```

self reference lets us match arbitrarily long lists

Part B – Function operations on List

① Recursive calls (call itself)

```
(B) Design a function that consumes ListOfString and produces true if
    the list includes "UBC".

;; ListOfString is one of:
;;  - empty
;;  - (cons String ListOfString)
;; interp. a list of strings
(define LOS1 empty)
(define LOS2 (cons "McGill" empty))
(define LOS3 (cons "UBC" (cons "McGill" empty)))
#;
(define (fn-for-los los)
  (cond [(empty? los) (...)]
        [else
         (... (first los)
              (fn-for-los (rest los)))]))

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: empty
;;  - compound: (cons String ListOfString)
;;  - self-reference: (rest los) is ListOfString

;; ListOfString -> Boolean
;; produce true if los includes "UBC"
(check-expect (contains-ubc? empty) false)
(check-expect (contains-ubc? (cons "McGill" empty)) false)
(check-expect (contains-ubc? (cons "UBC" empty)) true)
(check-expect (contains-ubc? (cons "McGill" (cons "UBC" empty))) true)

;(define (contains-ubc? los) false) ;stub
```

SR (self-reference)

natural recursion

An arbitrary amount of information

An arbitrary amount of information

Well formed self-reference:
- at least one base case
- at least one self reference case

Examples should include base and self-reference cases.

The self-reference template rule puts a natural recursion in the template that corresponds to the self-reference in the type comment.

② Design with Lists

```
(define (sum lon)
  (cond [(empty? lon) 0]
        [else
         (+ (first lon)
            (sum (rest lon)))]))

(define (count lon)
  (cond [(empty? lon) 0]
        [else
         (+ 1
            (count (rest lon)))]))

(define (contains-ubc? los)
  (cond [(empty? los) false]
        [else
         (if (string=? (first los) "UBC")
             true
             (contains-ubc? (rest los)))]))
```

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

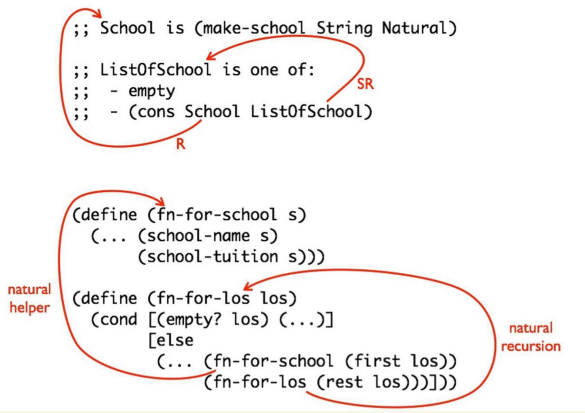
how contribution of first and result of natural recursion are combined

	base	contribution of first	combination
sum	0	itself	+
count	0	1	+
contains-ubc?	false	(string=? <x> "UBC")	(if <y> true <nr>)

Lecture 6 – Reference

Part A – The Reference Rule

① Helper

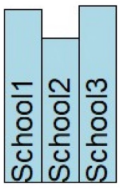


② for complicated questions & function, write test examples first is important.

(functional language benefits a lot from copy & paste)

③ Explanations with "schools bar" example

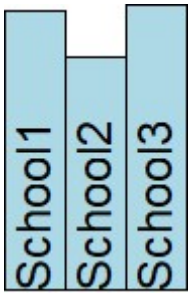
```
;
PROBLEM:

Eva is trying to decide where to go to university. One important factor for her is
tuition costs. Eva is a visual thinker, and has taken Systematic Program Design,
so she decides to design a program that will help her visualize the costs at
different schools. She decides to start simply, knowing she can revise her design
later.

The information she has so far is the names of some schools as well as their
international student tuition costs. She would like to be able to represent that
information in bar charts like this one:
```




```
(A) Design data definitions to represent the information Eva has.
(B) Design a function that consumes information about schools and their
    tuition and produces a bar chart.
(C) Design a function that consumes information about schools and produces
    the school with the lowest international student tuition.
```

Codes in next 2 pages

```
(require 2htdp/image)

;; Constants:

(define FONT-SIZE 24)
(define FONT-COLOR "black")

(define Y-SCALE   1/200)
(define BAR-WIDTH 30)
(define BAR-COLOR "lightblue")
;===========================================
;; Data definitions:

(define-struct school (name tuition))
;; School is (make-school String Natural)
;; interp. name is the school's name, tuition is international-students tuition in
USD

(define S1 (make-school "School1" 27797)) ;We encourage you to look up real schools
(define S2 (make-school "School2" 23300)) ;of interest to you -- or any similar
data.
(define S3 (make-school "School3" 28500)) ;

(define (fn-for-school s)
  (... (school-name s)
       (school-tuition s)))

;; Template rules used:
;;  - compound: (make-school String Natural)


;; ListOfSchool is one of:
;;  - empty
;;  - (cons School ListOfSchool)
;; interp. a list of schools
(define LOS1 empty)
(define LOS2 (cons S1 (cons S2 (cons S3 empty))))

(define (fn-for-los los)
  (cond [(empty? los) (...)]
        [else
         (... (fn-for-school (first los))
              (fn-for-los (rest los)))]))

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: empty
;;  - compound: (cons School ListOfSchool)
;;  - reference: (first los) is School
;;  - self-reference: (rest los) is ListOfSchool

;===========================================


;; Functions:
;
```



```
;; ListOfSchool -> Image
;; produce bar chart showing names and tuitions of consumed schools
(check-expect (chart empty) (square 0 "solid" "white"))
(check-expect (chart (cons (make-school "S1" 8000) empty))
                     (beside/align "bottom"
                                   (overlay/align "center" "bottom"
                                                  (rotate 90 (text "S1" FONT-SIZE
FONT-COLOR))
                                                  (rectangle BAR-WIDTH (* 8000 Y-SCALE)
"outline" "black")
                                                  (rectangle BAR-WIDTH (* 8000 Y-SCALE)
"solid" BAR-COLOR))
                                   (square 0 "solid" "white")))

(check-expect (chart (cons (make-school "S2" 12000) (cons (make-school "S1" 8000)
empty)))
                     (beside/align "bottom"
                                   (overlay/align "center" "bottom"
                                                  (rotate 90 (text "S2" FONT-SIZE
FONT-COLOR))
                                                  (rectangle BAR-WIDTH (* 12000 Y-SCALE)
"outline" "black")
                                                  (rectangle BAR-WIDTH (* 12000 Y-SCALE)
"solid" BAR-COLOR))
                                   (overlay/align "center" "bottom"
                                                  (rotate 90 (text "S1" FONT-SIZE
FONT-COLOR))
                                                  (rectangle BAR-WIDTH (* 8000 Y-SCALE)
"outline" "black")
                                                  (rectangle BAR-WIDTH (* 8000 Y-SCALE)
"solid" BAR-COLOR))
                                   (square 0 "solid" "white")))

;(define (chart los) (square 0 "solid" "white")) ;stub

(define (chart los)
  (cond [(empty? los) (square 0 "solid" "white")]
        [else
         (beside/align "bottom"
                       (make-bar (first los))
                       (chart (rest los)))]))

;; School -> Image
;; produce the bar for a single school in the bar chart
(check-expect (make-bar (make-school "S1" 8000))
              (overlay/align "center" "bottom"
                             (rotate 90 (text "S1" FONT-SIZE FONT-COLOR))
                             (rectangle BAR-WIDTH (* 8000 Y-SCALE) "outline"
"black")
                             (rectangle BAR-WIDTH (* 8000 Y-SCALE) "solid"
BAR-COLOR)))

;(define (make-bar s) (square 0 "solid" "white")); stub

(define (make-bar s)
   (overlay/align "center" "bottom"
                  (rotate 90 (text (school-name s) FONT-SIZE FONT-COLOR))
                  (rectangle BAR-WIDTH (* (school-tuition s) Y-SCALE) "outline"
"black")
                  (rectangle BAR-WIDTH (* (school-tuition s) Y-SCALE) "solid"
BAR-COLOR)))
```

Lecture 7 - Naturals

Part A — Natural Numbers

① "add1"/"sub1" primitive operation : takes a natural number & add/substract 1.

② design our version of natural numbers (a list 0~9 only) & operations from scratch

```
;; NATURAL is one of:
;;  - empty
;;  - (cons "!" NATURAL)
;; interp. a natural number, the number of "!" in the list is the number
(define N0 empty)
(define N1 (cons "!" N0)) ;1
(define N2 (cons "!" N1)) ;2
(define N3 (cons "!" N2))
(define N4 (cons "!" N3))
(define N5 (cons "!" N4))
(define N6 (cons "!" N5))
(define N7 (cons "!" N6))
(define N8 (cons "!" N7))
(define N9 (cons "!" N8))

;; These are the primitives that operate NATURAL:
(define (ZERO? n) (empty? n))
(define (ADD1 n) (cons "!" n))
(define (SUB1 n) (rest n))
#;
(define (fn-for-NATURAL n)
  (cond [(ZERO? n) (...)]
        [else
         (...n
              (fn-for-NATURAL (SUB1 n)))]))
;=========================================
;; NATURAL NATURAL -> NATURAL
;; produce a + b
(check-expect (ADD N2 N0) N2)
(check-expect (ADD N0 N3) N3)
(check-expect (ADD N3 N4) N7)

;(define (ADD a b) N0) ;stub

(define (ADD a b)
  (cond [(ZERO? b) a]
        [else
         (ADD (ADD1 a) (SUB1 b))]))
;=====================================
;; NATURAL NATURAL -> NATURAL
;; produce a - b
(check-expect (SUBSTRACT N2 N0) N2)
(check-expect (SUBSTRACT N6 N2) N4)

;(define (SUBSTRACT a b) N0) ;stub

(define (SUBSTRACT a b)
  (cond [(ZERO? b) a]
        [else
         (SUBSTRACT (SUB1 a) (SUB1 b))]))
```

Lecture 8 – Helpers

Part A – Function Composition

① Function Composition : when a function must perform two or more distinct and complete operations on the consumed data, e.g. sort + layout a list of images

(ensure testing the composed + separate functions)

② "Arbitrary # of images" example

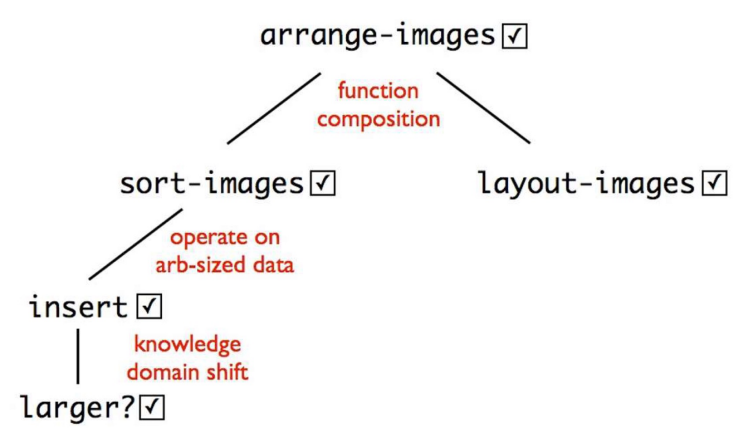
Part B – Operating on a List

① sort & insert in a list – design our version in the "Arbitrary # of images" example.

Part C – Domain Knowledge Shift

① the example involves knowledge domain shift from inserting into a list to comparing size of images. This caused us to use a helper

Part D – Summary of the Example structure



Codes in next 2 pages

```
(require 2htdp/image)

;
 PROBLEM:

 In this problem imagine you have a bunch of pictures that you would like to
 store as data and present in different ways. We'll do a simple version of that
 here, and set the stage for a more elaborate version later.

 (A) Design a data definition to represent an arbitrary number of images.

 (B) Design a function called arrange-images that consumes an arbitrary number
     of images and lays them out left-to-right in increasing order of size.


;; Constants:

(define BLANK (square 0 "solid" "white"))

;; for testing:
(define I1 (rectangle 10 20 "solid" "blue"))
(define I2 (rectangle 20 30 "solid" "red"))
(define I3 (rectangle 30 40 "solid" "green"))

;; Data definitions:

;; ListOfImage is one of:
;;  - empty
;;  - (cons Image ListOfImage)
;; interp. An arbitrary number of images
(define LOI1 empty)
(define LOI2 (cons I1
                   (cons I2
                         empty)))
#;
(define (fn-for-loi loi)
  (cond [(empty? loi) (...)]
        [else
         (... (first loi)
              (fn-for-loi (rest loi)))]))

;; Functions:

;; ListOfImage -> Image
;; lay out images left to right in increasing order of size
;; sort images in increasing order of size and then lay them out left-to-right
(check-expect (arrange-images (cons I1 (cons I2 empty)))
              (beside I1 I2 BLANK))
(check-expect (arrange-images (cons I2 (cons I1 empty)))
              (beside I1 I2 BLANK))

;(define (arrange-images loi) BLANK) ;stub

(define (arrange-images loi)
  (layout-images (sort-images loi)))


;; ListOfImage -> Image
;; place images beside each other in order of list
(check-expect (layout-images empty) BLANK)
(check-expect (layout-images (cons I1 (cons I2 empty)))
              (beside I1 I2 BLANK))

;(define (layout-images loi) BLANK) ;stub

(define (layout-images loi)
  (cond [(empty? loi) BLANK]
```

```
        [else
         (beside (first loi)
                 (layout-images (rest loi)))]))

;; ListOfImage -> ListOfImage
;; sort images in increasing order of size (area)
(check-expect (sort-images empty) empty)
(check-expect (sort-images (cons I1 (cons I2 empty)))
              (cons I1 (cons I2 empty)))
(check-expect (sort-images (cons I2 (cons I1 empty)))
              (cons I1 (cons I2 empty)))
(check-expect (sort-images (cons I3 (cons I1 (cons I2 empty))))
              (cons I1 (cons I2 (cons I3 empty))))

;(define (sort-images loi) loi)

(define (sort-images loi)
  (cond [(empty? loi) empty]
        [else
         (insert (first loi)
                 (sort-images (rest loi)))])) ;result of natural recursion will be
sorted

;; Image ListOfImage -> ListOfImage
;; insert img in proper place in loi (in increasing order of size)
;; ASSUME: loi is already sorted
(check-expect (insert I1 empty) (cons I1 empty))
(check-expect (insert I1 (cons I2 (cons I3 empty))) (cons I1 (cons I2 (cons I3
empty))))
(check-expect (insert I2 (cons I1 (cons I3 empty))) (cons I1 (cons I2 (cons I3
empty))))
(check-expect (insert I3 (cons I1 (cons I2 empty))) (cons I1 (cons I2 (cons I3
empty))))

;(define (insert img loi) loi) ;stub

(define (insert img loi)       ;two parameters
  (cond [(empty? loi) (cons img empty)]
        [else
         (if (larger? img (first loi))
             (cons (first loi)
                   (insert img
                           (rest loi)))
             (cons img loi))]))

;; Image Image -> Boolean
;; produce true if img1 is larger than img2  (by area)
(check-expect (larger? (rectangle 3 4 "solid" "red") (rectangle 2 6 "solid"
"red")) false)
(check-expect (larger? (rectangle 5 4 "solid" "red") (rectangle 2 6 "solid"
"red")) true)
(check-expect (larger? (rectangle 3 5 "solid" "red") (rectangle 2 6 "solid"
"red")) true)
(check-expect (larger? (rectangle 3 4 "solid" "red") (rectangle 5 6 "solid"
"red")) false)
(check-expect (larger? (rectangle 3 4 "solid" "red") (rectangle 2 7 "solid"
"red")) false)

;(define (larger? img1 img2) true) ;stub

(define (larger? img1 img2)
  (> (* (image-width img1) (image-height img1))
     (* (image-width img2) (image-height img2))))
```

Lecture 9 - Binary Search Tree

Part A - BSL with List Abbreviations

① Beginning Student with List Abbreviations: "list" keyword

e.g. > (list "a" "b" "c") (≡ > (cons "a" (cons "b" (cons "c" empty))))

② Careful: use "cons <oneElement> L1" to add L1 after the first element

use "append L1 L2" to concatenate the two lists together

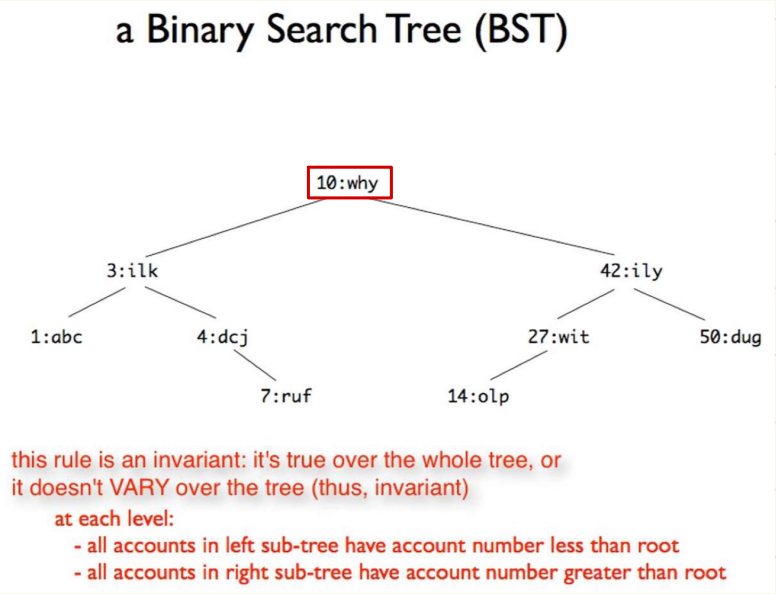
③ "List of Accounts" with List and compound data "define-struct account (num name)"

Part B - Binary Search Trees

① looking up member names in the "List of Accounts" – how to speed up?

sort first, then binary search

② a Binary Search Tree for the "List of Accounts" [(1:abc), (3:ilk), ..., (50:dug)]



each time the tree gets smaller by half

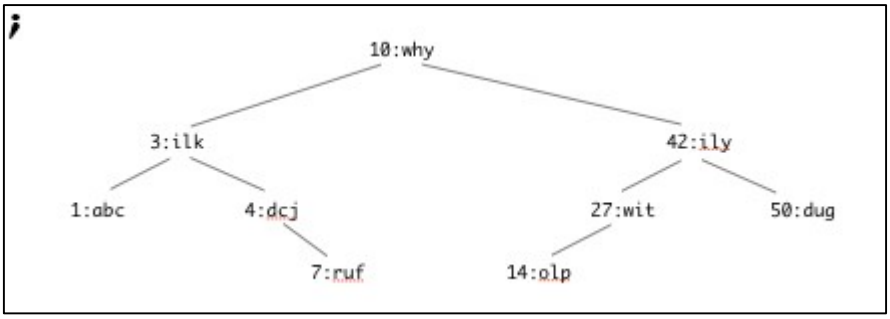
③ for "List of Accounts" nodes, 4 information can construct a node. Therefore, define node as compound data with the node's "key", "value", "left BST", "right BST"

④ Build a "lookup-key" function for the Binary Search Tree.

Codes in next 2 pages

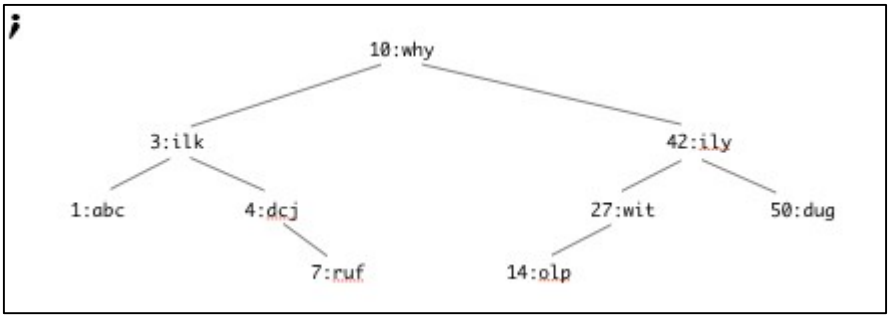
```
;; lookup-in-bst-starter.rkt

#;
 Consider the following data definition for a binary search tree:

;; Data definitions:

(define-struct node (key val l r))
;; A BST (Binary Search Tree) is one of:
;;  - false
;;  - (make-node Integer String BST BST)
;; interp. false means no BST, or empty BST
;;         key is the node key
;;         val is the node val
;;         l and r are left and right subtrees
;; INVARIANT: for a given node:
;;     key is > all keys in its l(eft)  child
;;     key is < all keys in its r(ight) child
;;     the same key never appears twice in the tree
```

#;




```
(define BST0 false)
(define BST1 (make-node 1 "abc" false false))
(define BST4 (make-node 4 "dcj" false (make-node 7 "ruf" false false)))
(define BST3 (make-node 3 "ilk" BST1 BST4))
(define BST42
  (make-node 42 "ily"
             (make-node 27 "wit" (make-node 14 "olp" false false) false)
             (make-node 50 "dug" false false)))
(define BST10
  (make-node 10 "why" BST3 BST42))
#;
(define (fn-for-bst t)
  (cond [(false? t) (...)]
        [else
         (... (node-key t)    ;Integer
              (node-val t)    ;String
              (fn-for-bst (node-l t))
              (fn-for-bst (node-r t)))]))

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic-distinct: false
;;  - compound: (make-node Integer String BST BST)
;;  - self reference: (node-l t) has type BST
;;  - self reference: (node-r t) has type BST

;; Functions:
```

```
;PROBLEM:

 Complete the design of the lookup-key function below. Note that because this is a s
 it will sometimes 'fail'. This happens if it is called with an key that does not ex
 it is provided. If this happens the function should produce false. The signature fo
 is written in a special way as shown below.
```




```
;; BST Natural -> String or false
;; Try to find node with given key, if found produce value; if not found produce
false.
(check-expect (lookup-key BST0  99) false)

(check-expect (lookup-key BST1   1) "abc")
(check-expect (lookup-key BST1   0) false) ;L fail
(check-expect (lookup-key BST1  99) false) ;R fail
(check-expect (lookup-key BST10  1) "abc") ;L L succeed
(check-expect (lookup-key BST10  4) "dcj") ;L R succeed
(check-expect (lookup-key BST10 27) "wit") ;R L succeed
(check-expect (lookup-key BST10 50) "dug") ;R R succeed

;(define (lookup-key t k) "")

;<template according to BST, and additional atomic parameter k>

(define (lookup-key t k)
  (cond [(false? t) false]
        [else
         (cond [(= k (node-key t)) (node-val t)]
               [(< k (node-key t))   ;should we go left?
                     (lookup-key (node-l t) k)]
               [(> k (node-key t))   ;should we go right?
                     (lookup-key (node-r t) k)])]))
```