TeachScheme! 2: Boston: 5–9 July 2004

`http://www.ccs.neu.edu/home/matthias/TS2-2004/`

This hand out consists of two pieces per daily session: goals and examples for the lecture session and exercises for the lab session. Please read the lab instructions carefully.

Lab instructions:

1. Please login as "pclab" with the password "guest". Use the "Log on to:" box and choose the domain "CCIS WINDOWS", though this may already be the case.

2. Create a Desktop folder for your work. The folder is likely to stay around for the week, but due to open access for a couple of hours in the evening, a misinformed user may delete them. Back up the folder on one of the floppies from your welcome folder.

3. For the lab sessions, please scan the exercises quickly. Solve those exercises that are marked with an asterisk (*). Then decide which exercise you like most; try to do complete that one. Repeat this until you are out of time.

Contents

Goals of the Workshop

We have two goals with this workshop:

1. We wish to deepen your understanding of program design, especially structural recursion, abstraction, generative recursion, and the model-view "software architecture".

2. We try to give you a sense of how program design à la HtDP applies to practical programming. This year we have chosen three topics: games, web programming, and network scripting.

The Daily Routine

The workshop schedule is roughly like the one you know from TeachScheme! 1. Every morning we start at 8:30am with a lecture session followed by a lab. The lunch break is from noon to 1:15pm. The afternoon starts again with a brief lecture session, followed by more lab time. The goal is to wrap up each day by 5:00pm.

1 Monday Morning: Welcome Back

Related material in *How to Design Programs*: parts I, II

1.1 Goals

(1) recall basic Scheme data: numbers, images, structures, lists

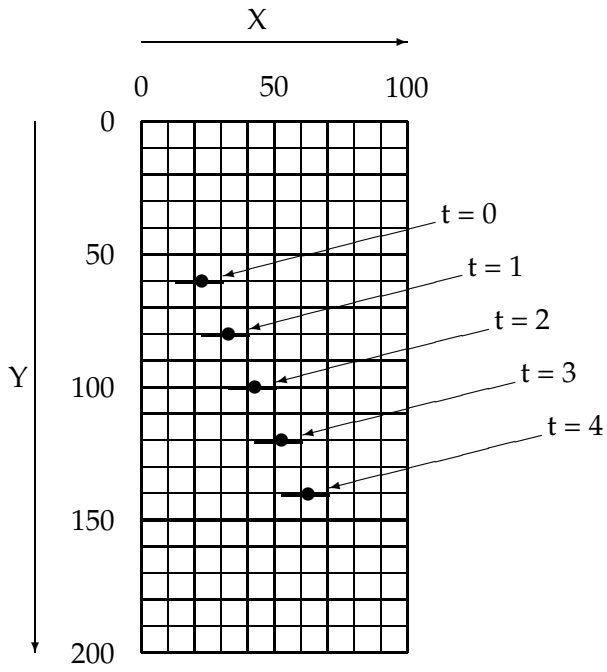
(2) recall basic Scheme mechanics: defining functions, structures, variables

(3) recall the design recipes for structural data:

1. analyze the problem & identify the classes of data (with examples)
2. formulate a purpose statement and a contract
3. make up "functional" examples
4. create the template: what information exists to compute the desired result?
5. program
6. turn the examples into a test suite

(4) over the week, see iterative refinement at work

1.2 Example: Functions on Numbers



Translate into a function from numbers to numbers:

$t =$	0	1	2	3	4	...	7	
$X =$	20	30	40	50	60	...	?	

```
;; ufo-x : Number → Number
;; where is the UFO at t (horizontally)?
;; examples: see table
(define (ufo-x t)
  (+ 20 (* 10 t)))

;; Tests:
(= (ufo-x 0) 20)
(= (ufo-x 1) 30)
(= (ufo-x 2) 40)

;; Run program run:
(ufo-x 2345)
```

1.3 Example: Images

```
(define an-image (rectangle 3 4 'blue))

;; area : Image → Number
;; compute the number of pixels in an image
;; example: (area an-image) should be 12
(define (area i)
  (* (image-width i) (image-height i)))

;; Tests:
(= (area an-image) 12)

;; Run program run: (demo matthias demo)
(area ... an image of your favorite teach-scheme instructor ...)
```

1.4 Example: Structures and Functions on Structures

```
(define-struct color (red green blue)) ;; color occurs in nature
;; Color = (make-color Number Number Number)

;; make-color; color?; color-red, color-green, color-blue

#| Template:
;; Color → ???
(define (process-rgb c)
  ... (color-red c) ...
  ... (color-green c) ...
  ... (color-blue c) ... )
|#

;; Program:
;; Color → Color
;; produce a color like c, but without any green
;; example: (remove-green (make-color 200 100 0)) should be
;; (remove-green (make-color 200 0 0))
(define (remove-green c)
  (make-color (color-red c) 0 (color-blue c)))

;; Tests:
(equal? (remove-green (make-color 200 100 0))
        (make-color 200 0 0))
```

1.5 Example: Lists and Functions on Lists

```
;; A ColorList is either
;; — empty
;; — (cons Color ColorList)

#| Template:
;; ColorList → ???
(define (process-cl cl)
  (cond
    [(empty? cl) ...]
    [else ... (first cl) ... (process-cl (rest cl)) ...]))
|#

;; ColorList → ColorList
;; remove green from every color on cl
;; examples: see tests
(define (remove-all-green cl)
  (cond
    [(empty? cl) empty]
    [else (cons (remove-green (first cl)) (remove-all-green (rest cl)))]))

;; Tests:
(equal? (remove-all-green (cons (make-color 100 100 100)
                                (cons (make-color 200 100 0)
                                      empty)))
        (cons (make-color 100 0 100)
              (cons (make-color 200 0 0)
                    empty)))

;; Run program run:
(color-list→image
  (remove-all-green
    (image→color-list
      ... an image of your favorite teach-scheme instructor ...)))
```

Exercises

Use the language level
Beginning Student

Exercise 1.1 (*) Locate the Web page for the workshop (see cover). Bookmark it. Copy and paste the program for removing green (section 1.5) from a picture and run the tests. Then get the expression at the bottom of the program to run. Hint: Use DrScheme's help desk to find out how the primitives on images work. Yes, you may use any picture that you like and you have access to the Web. ∎

Exercise 1.2 (*) Strings are another important class of data that most programming languages offer. With number→string, you can even convert any number into a string, and with string→number you can convert a string into a number, if the string represents one. Experiment with the two functions in DrScheme's Interactions window.

Develop the function *numbers-to-strings*, which consumes a list of numbers and produces a list of strings that represent these numbers. For example, (cons 1 (cons 2 (cons 3 empty))) produces '("1" "2" "3").

Also develop the function *strings-to-numbers*, which consumes a list of strings and converts them to numbers, if possible. In other words, it uses string→number on each item on the list.

Finally, develop the program *strings-to-numbers∗*. It consumes a list of strings and converts them to numbers. If a string doesn't represent a number, it drops the item from the list. ∎

Exercise 1.3 Networks send requests. Servers process these requests and send responses back.

One way to represent a class of requests is to form a list where each item is a list of two items: a string and a boolean:

;; *Request* is either
;; — empty
;; — (cons *Line Request*)
;; *Line* is (cons *String* (cons *Boolean* empty))

Develop the function *which-toppings*. The function consumes a *Request* and produces the list of strings that come with true in the given request. ∎

2 Monday Afternoon: Images, Lists, N

Related material in *How to Design Programs*: intermezzo 2, part II[11]

2.1 Goals

(1) to make working with lists and S-expressions easier
(2) to get to know the class of image values
(3) to understand functions on *natural numbers*

2.2 A Notation for Lists

```
(cons 1
  (cons 2
    (cons 3 empty)))
```

```
(list 1 2 3)
```

```
(cons
  (cons 'a (cons 'b empty))
  (cons
    (cons 1 (cons 2 empty))
    (cons
      (cons true (cons false empty))
      (cons
        (cons "a" (cons "b" empty))
        empty))))
```

```
(list (list 'a 'b)
      (list 1 2)
      (list true false)
      (list "a" "b"))
```

```
(cons (make-color 0 0 0)
  (cons (make-color 255 255 255)
    (cons (make-color 200 100 0)
      empty)))
```

```
(list (make-color 0 0 0)
      (make-color 255 255 255)
      (make-color 200 100 0))
```

2.3 Example: N

```
;; An N (natural number) is either
;; — 0
;; — (add1 N)
```

Recognize 0 with zero?. Given *n* and assuming (zero? *n*) is false, then *n* is (add1 *m*) for some *m*, which belongs to **N**. We get *m* via (sub1 *n*).

```
#| Template:
;; N → ???
(define (process-n k)
  (cond
    [(zero? k) ...]
    [else ... (process-n (sub1 k)) ...]))
|#

;; N Symbol → ListOfSymbols
;; make a list with k copies of a
;; examples:
;; — given 0 and 'z, produce empty
;; — given 2 and 'z, produce (list 'z 'z)
(define (copies n a)
  (cond
    [(zero? n) empty]
    [else (cons a (copies (sub1 n) a))]))

;; Tests:
(equal? (copies 0 'z) empty)
(equal? (copies 2 'z) (list 'z 'z))

;; N → ListOfString
;; create a list of strings "k", ..., "0"
(define (cells k)
  (cond
    [(zero? k) (list "0")]
    [else (cons (number→string k) (cells (sub1 k)))]))

;; Tests:
(equal? (cells 0) (list "0"))
(equal? (cells 2) (list "2" "1" "0"))
```

2.4 Example: More on Images

Images have always been values in DrScheme. All we had, however, were constants. Now we also have functions for creating images:

(rectangle 10 20 'red)	create a red, 10 x 20 rectangle
(circle 10 'red)	create a red circle with radius 10
(disk 10 'red)	create a red disk of radius 10
(empty-scene 100 200)	create an empty 100 x 200 scene
(place-image (rectangle 5 5 'red) 10 20 (empty-scene 100 200))	place a red, 5 x 5 square at (10,20) into an empty scene
(image→color-list (rectangle 3 4 'green))	turn the green, 3 x 4 reactangle into a list of 12 copies of (make-color 60 248 52)

```
(color-list→image
  (list
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52))
  1 12)
```

turn a list of 12 copies of (make-color 60 248 52) into a horizontal line of 12 green pixels

---

```
(color-list→image
  (list
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52)
    (make-color 60 248 52))
  3 4)
```

turn a list of 12 copies of (make-color 60 248 52) into a horizontal line of 12 green pixels

---

```
(offset-image+
  (rectangle 1 11 'red)
  −5 5
  (rectangle 11 1 'red))
```

create two thin rectangles, then combine them into a red cross

2.5 Example: Image Functions

```
(define UFO ...) ;; some shape that resembles a green saucer

;; Number → Number
;; determine the UFO's x coordinate at time t
(define (ufo-x t) 10)

;; Number → Number
;; determine the UFO's y coordinate at time t
(define (ufo-y t) (+ 20 t))

;; Number Scene → Scene
;; place the UFO into s at time t
(define (place-ufo t s) (place-image UFO (ufo-x t) (ufo-y t) s))

;; Tests:
(= (ufo-x (random 100)) 10)
(= (ufo-y 10) 30)
```

Use the stepper to see how things work.

Exercises

Use the language level
BS w/ Lists

Exercise 2.1 (*) DrScheme doesn't provide the function *line* for drawing a line. Fortunately, this is not a problem because we have color-list→image, which consumes a list of colors and produces an image.

Develop the function *make-line*, which creates a *ColorList* that represents a horizontal line. The function consumes the width of the line (a natural number) and a color. It produces an appropriate list of colors.

Also, even if DrScheme did not provide rectangle, we could still draw rectangles. Develop the function *make-rectangle*. It consumes two natural numbers, *width* and *height*, and a color. Its result is a ColorList that represents the matching rectangle.

Hint: (append (list 1 2) (list 3 4 5)) produces (list 1 2 3 4 5).

After you have developed the functions, including automated tests, run them as follows:

```
(color-list→image (make-line 100 (make-color 100 100 100)) 100 1)
```

or

```
(color-list→image (make-rectangle 2 3 (make-color 0 0 0)) 2 3)
```

Keep in mind that these are *not* tests. ∎

Add the teachpack
`image.ss`

Exercise 2.2 (*) Copy and paste the program from section 2.5. Provide a definition for *UFO* (in the lecture code) so that it shows up as a flying green saucer. ∎

Add the teachpack
`image.ss`

Exercise 2.3 (*difficult, but entertaining*) Develop *place-blocks*. The program consumes a list of blocks:

```
;; A BlockList is either:
;; — empty
;; — (cons N BlockList)
```

Each number on a *BlockList* represents the height of a stack of blocks. The program produces an image, drawing a stack of blocks of the specified height starting in the *lower, right corner* of a 100 x 100 canvas.

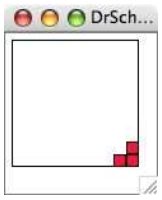
Note: The program is a core piece of a Tetris-style game.

Hint 1: Use these definitions for your basic constants:

```
(define WIDTH 100)
(define HEIGHT 100)
(define CANVAS (empty-scene WIDTH HEIGHT))
```

```
(define BLOCK-WIDTH 10)
(define BLOCK
  (offset-image+
   (rectangle BLOCK-WIDTH  BLOCK-WIDTH 'red)
   0 0
   (outline-rect BLOCK-WIDTH  BLOCK-WIDTH 'black)))
```

Hint 2: Here is the image for (*place-blocks* (list 1 2)):



Produce the image by hand and then use it with image=? to *test* the program. The picture shows that the drawing starts at the right. The length of the list determines how far from the right a stack of blocks is drawn. ∎

Exercise 2.4 (*) Use the conventional **draw.ss** teachpack to draw and erase green flying saucers from a canvas. That is, develop the functions *draw-ufo* and *clear-ufo* for drawing and clearing a UFO-like shape from a canvas. Both functions consume a *Posn*, which represents the anchor position of the UFO (e.g., the center of the disk or the northwest corner of the rectangle). ∎

Add the teachpack
`draw.ss`

3 Tuesday Morning: Local Definitions

Related material in *How to Design Programs*: intermezzo 3

3.1 Goals

(1) to study:
(1a) a new programming construct, **local** definitions,
(1b) and a related programming concept, lexical scope;
(2) how to approach new constructs: syntax, semantics, pragmatics

3.2 Local Definitions: Syntax and Semantics

Syntax (aka Grammar):

(**local** ((**define** *a-name rhs-expression*)
  ...
  (**define** *another-name another-rhs-expression*))
the-expression)

Syntax example:

(**local** ((**define** x 10)
  (**define** y ($+$ x 10)))
($+$ x ($*$ 10 y)))

Semantics (aka meaning): The evaluation of a **local** expression proceeds in stages. It begins with the evaluation of the first definition. The evaluation proceeds just like for a normal, aka top-level, definition. Once we know the value of the *rhs-expression*, *a-name* stands for this value *inside the* **local** *expression*. After all definitions are evaluated, the evaluator reduces *the-expression* to a value. This last value is the result of the **local** expression.
Semantics example:

(**local** ((**define** x 10)
  (**define** y ($+$ x 10)))
($+$ x ($*$ 10 y)))

First, 10 is a value, so x stands for 10. Second, ($+$ x 10) accordingly evaluates to 20, and y represents 20. Finally, ($+$ x ($*$ 10 y)) evaluates to 210 because x is 10 and y is 20 inside this **local** expression. Hence, the result of the **local** expression is 210.

On to pragmatics.

3.3 Local Definitions: Go slow!

```
;; Posn → Number
;; determine the distance of p to the origin: sqrt((p_x)^2 + (p_y)^2)
;; example: (distance0 (make-posn 3 4)) is 5
(define (distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))

;; tests:
(= (distance0 (make-posn 3 4)) 5)
(= (distance0 (make-posn 12 5)) 13)
```

Here is a "slower" version:

```
;; Posn → Number
;; determine the distance of p to the origin: sqrt((p_x)^2 + (p_y)^2)
;; example: (distance0 (make-posn 3 4)) is 5
(define (distance0 p)
  (local ((define x (posn-x p))
          (define y (posn-y p))
          (define sqrx (sqr x))
          (define sqry (sqr y))
          (define sum (+ sqrx sqry)))
    (sqrt sum)))

;; tests:
(= (distance0 (make-posn 3 4)) 5)
(= (distance0 (make-posn 12 5)) 13)
```

3.4 Local Definitions: Go fast!

```
;; A NEL is one of:
;; — (cons Number empty)
;; — (cons Number NEL)


;; NEL → Number
;; find the largest number in l
;; examples: (list 1 2 3) produces 3
(define (largest l)
  (cond
    [(empty? (rest l)) (first l)]
    [else (cond
            [(> (first l) (largest (rest l))) (first l)]
            [else (largest (rest l))])]))


;; test:
(= (largest (list 1 2 3)) 3)


;; run program run


(largest (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)) ; produces
17


(largest
 (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23))
```

Use **local** definitions instead:

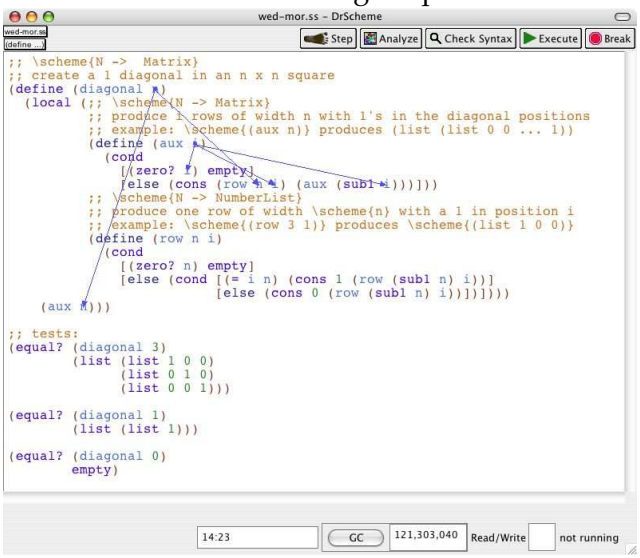
```
;; NEL → Number
;; find the largest number in l
;; examples: (list 1 2 3) produces 3
(define (largest l)
  (cond
    [(empty? (rest l)) (first l)]
    [else (local ((define max-in-rest (largest (rest l))))
            (cond [(> (first l) max-in-rest) (first l)]
                  [else max-in-rest]))]))
```

3.5 Local Definitions: Just Do It

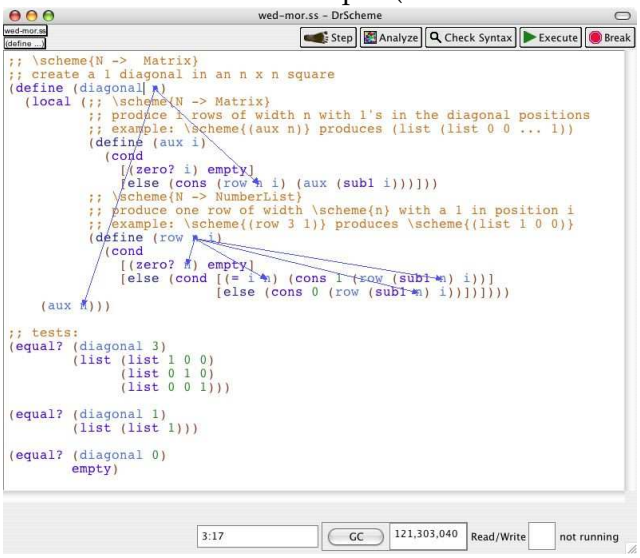
```
;; N → Matrix
;; create a 1 diagonal in an n x n square
(define (diagonal n)
  (local (;; N → Matrix
          ;; produce i rows of width n with 1's in the diagonal positions
          ;; example: (aux n) produces (list (list 0 0 ... 1))
          (define (aux i)
            (cond
              [(zero? i) empty]
              [else (cons (row n i) (aux (sub1 i)))]))
          ;; N → NumberList
          ;; produce one row of width n with a 1 in position i
          ;; example: (row 3 1) produces (list 1 0 0)
          (define (row n i)
            (cond
              [(zero? n) empty]
              [else (cond [(= i n) (cons 1 (row (sub1 n) i))]
                          [else (cons 0 (row (sub1 n) i))])])))
    (aux n)))


;; tests:
(equal? (diagonal 3)
        (list (list 1 0 0)
              (list 0 1 0)
              (list 0 0 1)))


(equal? (diagonal 1)
        (list (list 1)))


(equal? (diagonal 0)
        empty)
```

3.6 Lexical Scope

Identifier Bindings Exposed



Holes in Nested Lexical Scopes (Remember Pascal?)



Exercises

Use the language level
Intermediate Student

Exercise 3.1 (*) Take a look at this function definition:

```
;; Number → Number
;; compute the after-tax weekly wage from the hours someone worked
;; pay-per-hour: $12.55
;; tax rate: 15%
;; social security: 5.5%
;; insurance cost: $100
(define (after-tax-income hours)
  (- ; gross income:
     (* hours 12.55)
     ; taxes:
     (* (* hours 12.55) .15)
     ; social security:
     (* (* hours 12.55) .055)
     ; insurance
     100))

;; example/test:
(= (after-tax-income 40) 299.09)
```

Use **local** to simplify the function body ("go slow!"). ∎

Exercise 3.2 (*) Developing the program *strings-to-numbers∗* in exercise 1.2 required the definition of several functions. It is good practice to reorganize programs such as those with **local** into a single function definition. ∎

Exercise 3.3 Develop the program *numeric-matrix?*. The function consumes a string matrix. It produces true if all strings represent a number; otherwise its return value is false.

Here are the data definitions:

```
;; A Matrix is one of:
;; — empty
;; — (cons LOS Matrix)
;; Constraint: In a matrix, the length of all LOS is the same.

;; A LOS is one of:
;; — empty
;; — (cons String LON)
```

Hint: The program requires several function definitions.

Constraint: Develop the program in two stages. First design all functions at the top-level. Second organize them using **local** so that there is only one top-level function. ∎

Exercise 3.4 Design *make-spread*. The function consumes two natural numbers: n and m. Its result is a rectangular list: rectangle S-expression such as this:

```
(list
  (list (list 'td "cell00") (list 'td "cell01"))
  (list (list 'td "cell10") (list 'td "cell11"))
  (list (list 'td "cell20") (list 'td "cell21"))
  (list (list 'td "cell30") (list 'td "cell31")))
```

with 4 rows and 2 columns.

Hint: Use the function string-append, which performs the obvious operation on strings.

Use the function to develop the function *make-table*, which consumes two natural numbers and creates the following S-expression for $n = 3$ and $m = 2$:

```
(list 'table
  (list 'tr (list 'td "cell00") (list 'td "cell01"))
  (list 'tr (list 'td "cell10") (list 'td "cell11"))
  (list 'tr (list 'td "cell20") (list 'td "cell21")))
```

Add the teachpack
`servlet2.ss`

When the function is properly designed, use the function *inform/html* (from the teachpack) to display the result of the function in a browser. ∎

4 Tuesday Afternoon: Generative Recursion

Related material in *How to Design Programs*: part v

4.1 Goal

to study
(1) a new form of program organization and
(2) a design recipe for creating such programs

4.2 Example: Generating Balloons

Problem:

> Develop a function that creates a blue background with *n* randomly distributed balloons for an animated scene.

Or,

> Develop a function that creates a blue background with *n* randomly distributed falling stars for the "Collect Star Thalers" fairy-tale game.

We clearly need *n* distinct *Posn*s:

```
;; Number → PosnList
;; create a list of n distinct Posns
(define (background n)
  (cond
    [(zero? n) empty]
    [else (add-distinct-posn (background (sub1 n)))]))

;; PosnList → PosnList
;; create and add a Posn to l that is distinct from all
;; Posns on there
(define (add-distinct-posn l)
  (local ((define new-x (random WIDTH))
          (define new-y (random HEIGHT))
          (define new-p (make-posn new-x new-y)))
    (cond
      [(boolean? (member new-p l)) (cons new-p l)]
      [else (add-distinct-posn l)])))
```

What's wrong with the underlined recursion?

4.3 Example: Sorting Numbers

Problem:

> Sort a list of numbers via a divide-and-conquer strategy. That is, pick a pivot element; extract all those items that are smaller than the pivot and sort those; extract all those items that are larger than the pivot and sort those; and finally juxtapose the two sequences to get a sorted sequence.

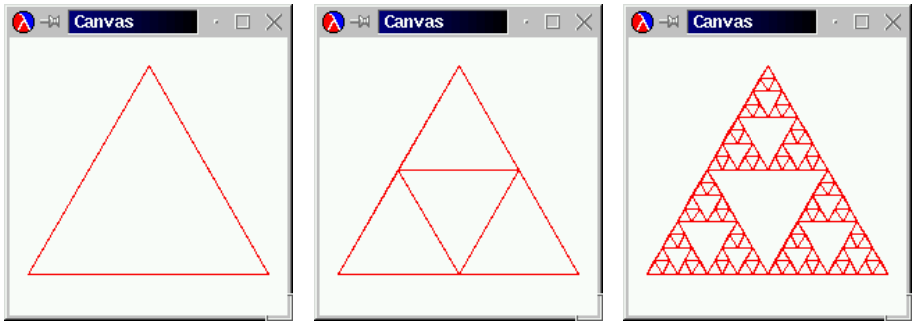
```
;; NumberList → NumberList
;; to create a sorted list of numbers from alon
(define (kwik-sort alon)
  (cond
    [(sorted? alon) alon]
    [else (append
            (kwik-sort (smaller (first alon) (rest alon)))
            (list (first alon))
            (kwik-sort (larger (first alon) (rest alon))))]))

;; NumberList → NumberList
;; extract all items from l that are smaller than pivot
(define (smaller pivot l)
  (cond
    [(empty? l) empty]
    [else (cond
            [(< (first l) pivot) (cons (first l) (smaller pivot (rest l)))]
            [else (smaller pivot (rest l))])]))

;; NumberList → NumberList
;; extract all items from l that are larger than pivot
(define (larger pivot l)
  (cond
    [(empty? l) empty]
    [else (cond
            [(> (first l) pivot) (cons (first l) (larger pivot (rest l)))]
            [else (larger pivot (rest l))])]))

;; test:
(equal? (kwik-sort (list 3 8 1 2 9 7)) (list 1 2 3 7 8 9))
```

What's wrong with the underlined recursions?

4.4 Example: Fractals

Problem:

Design a program that draws these triangles:




```
;; Posn Posn Posn → true
;; to draw a Sierpinski triangle at a, b, and c,
(define (sierpinski a b c)
  (cond
    [(too-small? a b c) #t]
    [else
      (local ((define a-b (mid-point a b))
              (define b-c (mid-point b c))
              (define c-a (mid-point a c)))
        (and
          (draw-triangle a b c)
          (sierpinski a a-b c-a)
          (sierpinski b a-b b-c)
          (sierpinski c c-a b-c)))]))

;; Posn Posn → Posn
;; to compute the mid-point between a-posn and b-posn
(define (mid-point a-posn b-posn)
  (make-posn (mid (posn-x a-posn) (posn-x b-posn))
             (mid (posn-y a-posn) (posn-y b-posn))))

;; Number Number → Number
;; to compute the average of x and y
(define (mid x y) (/ (+ x y) 2))
```

4.5 Generative Recursion: The Design Recipe

1. The design recipe uses the basic six steps.
   Question: if the ideas are *ad hoc*, how can we design a template systematically?

2. *All* functions based on generative recursion use the same basic outline. To find it, ask these four questions:

   (a) what are *trivial* solutions?

   (b) how do you generate new, slightly different problems?

   (c) how do you use the solutions to these problems?

   (d) how do you *now* to get the solution for the original problem?

3. We need a seventh step. Does it terminate?

Exercises

Exercise 4.1 (*) Develop the function *random-interval*. It consumes a number (*MIN*) and produces an interval:

```
(define-struct interval (left right))
;; An Interval is:
;; — (make-interval L R)
;; if (< L R) is true
```

The function returns an interval whose distance is less than *MIN*. ∎

Exercise 4.2 (*) Assume DrScheme's Definitions Window contains the definition for a mathematically continuous function f:

```
;; f : Number → Number
(define (f x) ...)
```

Just a reminder: continuous means for us "has no gaps."

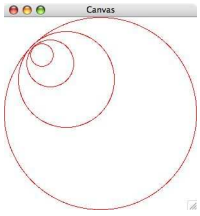
Design the function *find0*, which consumes an interval and determines a small interval (smaller than 1.0) where some top-level function f from *Number* to *Number* crosses the x-axis in this interval.

Hint 1: If f values at the two ends of the interval are at opposite sides of the x-axis, f is 0 somewhere in between. Take a look at the middle of the interval.

Hint 2: Your problem and data analysis may benefit from drawing a picture.

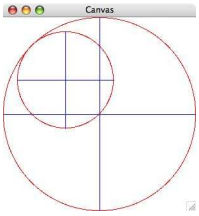
Note: The solution of this problem is the basis of binary search in vectors and similar data structures. ∎

Exercise 4.3 (*) Develop a program that draw nested circles like these:



The program consumes the center and radius of a circle. Place the initial circle in the center of the canvas created with (*start* 300 300).

Hint: Use the following picture to figure out the mathematics for finding the center of the next nested circle:



Exercise 4.4 Develop the function *merge-sort*, which sorts a list of numbers in ascending order using the following insight.

1. Construct a list of one-item lists from the given list of numbers. For example,

   ```
   (list 2 5 9 3)
   ```

   turns into

   ```
   (list (list 2) (list 5) (list 9) (list 3))
   ```

2. Merge pairs of neighboring, sorted lists in the list of list so that the result is sorted. For example,

   ```
   (list (list 2) (list 5) (list 9) (list 3))
   ```

   turns into

   ```
   (list (list 2 5) (list 3 9))
   ```

   and

   ```
   (list (list 2 5) (list 3 9))
   ```

   turns into

   ```
   (list (list 2 3 5 9))
   ```

In general, this step yields a list that is approximately half as long as the input. Why is the output not always half as long as the input?

3. Stop when the list of lists contains a single list. For example, for the above starting point, we should get

```
(list (list 2 3 5 9))
```

Develop all necessary functions independently. ∎

5 Wednesday Morning: Creating Abstractions

Related material in *How to Design Programs*: part iv

5.1 Goal

to abstract over recurring patterns in functions, why it matters
the design recipe

1. recognize patterns in two functions: circle the differences and connect them with lines in the two definitions;
2. for each pair of differences, introduce a new parameter and use it in place of the circled deltas;
3. define each of the old functions in terms of the abstraction and use the old test suite to ensure that things still work.

5.2 Example: Functions are Values, too

What's the difference?

```
;; draw-ufo : Posn → true
;; draw a red disk at p
(define (draw-ufo p)
  (draw-solid-disk p 3 'red))
```

```
;; clear-ufo : Posn → true
;; erase a red disk at p
(define (clear-ufo p)
  (clear-solid-disk p 3 'red))
```

Doing better:[1]

```
;; dc-ufo : Posn ??? → true
;; draw or erase a red disk at p
(define (clear-ufo p action)
  (action p 3 'red))
```

```
;; draw-ufo : Posn → true
;; draw a red disk at p
(define (draw-ufo p)
  (dc-ufo p draw-solid-disk))
```

```
;; clear-ufo : Posn → true
;; erase a red disk at p
(define (clear-ufo p)
  (dc-ufo p clear-solid-disk))
```

Now change the shape of the UFOs.

[1] There is a different alternative.

5.3 Example: Primitives are Values, too

```
;; NumberList → NumberList
;; extract all items from l
;; that are smaller than pivot
(define (smaller pivot l)
  (cond
    [(empty? l) empty]
    [else
      (cond
        [(< (first l) pivot)
         (cons (first l)
           (smaller pivot (rest l)))]
        [else
          (smaller pivot (rest l))])]))
```

```
;; NumberList → NumberList
;; extract all items from l
;; that are larger than pivot
(define (larger pivot l)
  (cond
    [(empty? l) empty]
    [else
      (cond
        [(> (first l) pivot)
         (cons (first l)
           (larger pivot (rest l)))]
        [else
          (larger pivot (rest l))])]))
```

```
;; test:
(equal? (smaller 3 (list 1 2 3 4 5))
        (list 1 2))
```

```
;; test:
(equal? (larger 3 (list 1 2 3 4 5))
        (list 4 5))
```

```
;; NumberList ??? → NumberList
;; extract all items from l
;; that are cmp to pivot
(define (extract pivot l cmp)
  (cond
    [(empty? l) empty]
    [else (cond
            [(cmp (first l) pivot)
             (cons (first l) (extract pivot (rest l)))]
            [else (extract pivot (rest l))])]))
```

```
(define (smaller pivot l)
  (extract pivot l <))
```

```
;; test:
(equal? (smaller 3 (list 1 2 3 4 5))
        (list 1 2))
```

```
(define (larger pivot l)
  (extract pivot l >))
```

```
;; test:
(equal? (larger 3 (list 1 2 3 4 5))
        (list 4 5))
```

Now extract all those numbers equal to pivot. Why?

5.4 Example: More on Functions

```
;; PosnList → NumberList
;; extract the x coordinates
;; from the Posns
(define (xxxs pl)
  (cond
    [(empty? pl) empty]
    [else
      (cons
        (posn-x (first pl))
        (xxxs (rest pl)))]))
```

```
;; ColorList → ColorList
;; remove green from every color on cl
;; see section 1.5
(define (remove-all-green cl)
  (cond
    [(empty? cl) empty]
    [else
      (cons
        (remove-green (first cl))
        (remove-all-green (rest cl)))]))
```

```
;; Test:
(equal?
 (xxxs
  (list
    (make-posn 3 4)
    (make-posn 12 5)
    (make-posn 17 20)
    (make-posn 22 1)))
 (list
   3
   12
   12
   22))
```

```
;; Test:
(equal? (remove-all-green
          (cons
            (make-color 100 100 100)
            (cons
              (make-color 200 100 0)
              empty)))
        (cons
          (make-color 100 0 100)
          (cons
            (make-color 200 0 0)
            empty)))
```

```
;; ???List ??? → ???List
;; use f on every item in l
(define (every l f)
  (cond
    [(empty? l) empty]
    [else (cons (f (first l)) (every (rest l) f))]))
```

```
(define (xxxs gl)
  (every gl one-gpa))
```

```
(define (remove-all-green cl)
  (every cl remove-green))
```

Now extract all *x*-coordinates from a list of *Posn*s.

5.5 Example: Contracts are People, too

A *list of GradeRs* is either

1. empty or
2. (cons *s l*) where
   (a) *s* is a gradeR and
   (b) *l* is a list of GradeRs.

A *list of Colors* is either

1. empty or
2. (cons *s l*) where
   (a) *s* is a Color and
   (b) *l* is a list of Colors.

Use (Listof X) instead.

5.6 Function have Contracts, too

We have:

;; *tax: Number → Number*

;; *xxxs: PosnList → NumberList*

;; *draw-ufo : Posn →* true

So it's not surprising that:

;; *draw-solid-disk : Posn Number ColorSymbol →* true

;; *< : Number Number → Boolean*

;; *remove-green : Color → Color*

;; *one-gpa : GradeR → GPA*

And therefore:

;; *dc-ufo : Posn* (*Posn Number ColorSymbol →* true) *→* true

;; *extract : NumberList* (*Number Number → Boolean*) *→ NumberList*

But what about *every*? (*Listof X*) (*X → Y*) *→* (*Listof Y*)

Exercises

Exercise 5.1 (*) Develop the function *allergic*. The function consumes a *Request* (see exercise 1.3 and produces the list of strings that are associated with false in the *Request*.
Compare *allergic* with *toppings* (also see exercise 1.3). Abstract the two functions into a single function.
Optional: Can you use the new function to extract all positive numbers from a list of numbers? If not, can you create a common abstraction? ∎

Exercise 5.2 (challenging!) Compare these two functions:

```
;; N → Image
;; add n squares to a
;; 100 x 100 empty canvas
(define (add-boxes n)
  (cond
    [(zero? n) (empty-scene 100 100)]
    [else
      (place-image
        (rectangle 1 1 'green)
        (random 100)
        (random 100)
        (add-boxes (sub1 n)))]))

;; tests:
(image=? (add-boxes 0)
         (empty-scene 100 100))
(image? (add-boxes 10))
```

```
;; N → Number
;; to add n to pi without using +

(define (add-to-pi n)
  (cond
    [(zero? n) pi]
    [else
      (add1
        (add-to-pi (sub1 n)))]))

;; tests:
(= (add-to-pi 0) pi)
(= (add-to-pi 3) (+ pi 3))
(= (add-to-pi 10) (+ pi 10))
```

Abstract the two functions into one. ∎

Exercise 5.3 Develop the function *blink*. It consumes a natural number n and a position and draws and clears a blinking rectangle at that position n times.

Add the teachpack
`draw.ss`

Hint 1: Use *sleep-for-a-while*.

Hint 2: Develop two auxiliary functions: one for drawing an object and one for clearing it.

Design a common abstraction for the two auxiliary functions. Then change the function so that it draws/clears a "lunar lander." Approximate

the lander with one small rectangle atop a large rectangle and two short, rectangular legs sticking out from the large rectangle. ∎

Exercise 5.4 Design *all-numbers*. The function determines whether all items in a list (of arbitrary values) are numbers.

Develop the function *all-symbols*, which determines whether all items in a list are symbols.

Abstract the two functions into a single function. Use the new function to define a function that determines whether all numbers on a list of numbers are positive. ∎

Exercise 5.5 (*) Compare the following two data definitions:

```
;; A NumberBT is one of:
;; — (make-leaf Number)
;; — (make-node NumberBT NumberBT)
;; A StringBT is one of:
;; — (make-leaf String)
;; — (make-node StringBT NumberBT)
```

Abstract them into a single data definition for binary trees. ∎

Exercise 5.6 Use *every* to define *numbers-to-strings* from exercise 1.2. ∎

6 Wednesday Afternoon: Using Abstractions

Related material in *How to Design Programs*: section 21.1

6.1 Goals: 1001 Loops

(1) to iterate: to apply an *action* to each item in a piece of compound data
(2) studying **lambda**: writing actions is easy with **lambda**

```
;; build-list : N (N → X) → (listof X)
;; to construct (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

;; filter : (X → boolean) (listof X) → (listof X)
;; to construct a list from all those items on alox for which p holds
(define (filter p alox) ...)

;; map : (X → Y) (listof X) → (listof Y)
;; to construct a list by applying f to each item on alox
;; that is, (map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
(define (map f alox) ...)

;; andmap : (X → boolean) (listof X) → boolean
;; to determine whether p holds for every item on alox
;; that is, (andmap p (list x-1 ... x-n)) = (and (p x-1) (and ... (p x-n)))
(define (andmap p alox) ...)

;; ormap : (X → boolean) (listof X) → boolean
;; to determine whether p holds for at least one item on alox
;; that is, (ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))
(define (ormap p alox) ...)
```

Figure 1: Some of Scheme's loops

6.2 Examples: Mapping

Given:

```
;; ColorList → ColorList
;; remove green from every color on cl
(define (remove-all-green cl)
  (cond
    [(empty? cl) empty]
    [else (cons (remove-green (first cl)) (remove-all-green (rest cl)))]))

;; Color → Color
;; produce a color like c, but without any green
(define (remove-green c)
  (make-color (color-red c) 0 (color-blue c)))

;; Tests:
(equal? (remove-all-green (cons (make-color 100 100 100)
                                (cons (make-color 200 100 0)
                                      empty)))
        (cons (make-color 100 0 100)
              (cons (make-color 200 0 0)
                    empty)))
```

Better:

```
;; ColorList → ColorList
;; remove green from every color on cl
(define (remove-all-green cl)
  (local ((define (remove-green c)
            (make-color (color-red c) 0 (color-blue c))))
    (map remove-green cl)))
```

6.3 mapping with lambda, the Ultimate

Best:

```
;; ColorList → ColorList
;; remove green from every color on cl
(define (remove-all-green cl)
  (map (lambda (c) (make-color (color-red c) 0 (color-blue c))) cl))
```

6.4 Example: Filtering

Given:

```
;; NumberList → NumberList
;; extract all items from l that are larger than pivot
(define (larger pivot l)
  (cond
    [(empty? l) empty]
    [else (cond
            [(> (first l) pivot) (cons (first l) (larger pivot (rest l)))]
            [else (larger pivot (rest l))])]))

;; test:
(equal? (larger 3 (list 3 8 1 2 9 7)) (list 8 9 7))
```

Better:

```
;; NumberList → NumberList
;; extract all items from l that are larger than pivot
(define (larger pivot l)
  (local ((define (compare-item-to-pivot item)
            (> item pivot)))
    (filter compare-item-to-pivot l)))
```

6.5 filtering with lambda, the Ultimate

Best:

```
;; NumberList → NumberList
;; extract all items from l that are larger than pivot
(define (larger pivot l)
  (filter (lambda (item) (> item pivot)) l))
```

6.6 lambda, the ultimate

Syntax: (**lambda** (*variable* ...) *expression*)

Syntax examples:
1:

(**lambda** (*i*) *i*)

2:

(**lambda** (*i j*) (> *i j*))

3:

(**define** *f* (**lambda** (*i*) (+ (∗ 10 *i*) 3))) ; cmp. with:
(**define** (*f i*) (+ (∗ 10 *i*) 3))

Semantics: **lambda** creates a function without name

Semantics examples: 1: consume a value and produce the very same value

(**lambda** (*i*) *i*)

2: consume two numbers; check whether the first is larger than the second

(**lambda** (*i j*) (> *i j*))

3: consume a number, turn into string, prefix with "cell"

(**lambda** (*i*) (string-append "cell" (number→string *i*)))

6.7 Example: Building Lists, with lambda

I want lists like (list 0 1 2 . . .)

```
(build-list n (lambda (i) i))
```

How about (list 3 4 5 . . .)?

```
(build-list n (lambda (i) (+ 3 i)))
```

And (list 1 4 9 . . .)?

```
(build-list n (lambda (i) (* i i)))
```

Can I do more than numbers?

```
(color-list→image
  (build-list
    50 (lambda (i)
         (make-color (* i 10)
                     (* i 5)
                     (* i 3))))
  50 1)
```

And nested lists?

```
(build-list
  10 (lambda (i)
       (build-list
         10 (lambda (j)
              (if (= i j) 1 0)))))
```

Exercises

Exercise 6.1 (*) Define *numbers-to-strings* from exercise 1.2 with one of the loops. ∎

Use the language level
IS w/ lambda

Exercise 6.2 Use build-list to design the function *count-down*, which consumes a natural number *n* and produces the list of numbers *n* ... 0. ∎

Exercise 6.3 (*) Define the functions *line* and *make-rectangle* (see exercise 2.1 for the original definitions) with loops. ∎

Exercise 6.4 Define *which-toppings* and *allergic* from exercises 1.3 and 5.1, respectively, with loops. ∎

Exercise 6.5 Use loops to define *numeric-matrix?* (see exercise 3.3). ∎

Exercise 6.6 (*) Use Scheme's loops to develop the function *build-rectangle*. It consumes two natural numbers: *n* and *m*. Its result is a rectanglular S-expression such as this:

```
(list
  (list 'tr (list 'td "cell00") (list 'td "cell01"))
  (list 'tr (list 'td "cell10") (list 'td "cell11"))
  (list 'tr (list 'td "cell20") (list 'td "cell21"))
  (list 'tr (list 'td "cell30") (list 'td "cell31")))
```

with 4 rows and 2 columns. ∎

Exercise 6.7 (*) Use foldr to define the function *add-blocks*, which consumes a list of *Posn*s and places squares at these positions into an empty scene. ∎

Exercise 6.8 Develop the function *draw-stars*, which consumes a natural number *n* and creates a blue canvas of 100 x 100 pixels with *n* yellow disks of radius 2. ∎

Add the teachpack
`draw.ss`

7 Thursday Morning: Some Basic I/O

7.1 Goal: Shriram's Day of Networking

to understand how Scheme programs can communicate with the rest of the world via sequential reading and writing

7.2 apply, another powerful loop

(+ 2 3)	add 2 and 3
(+ 2 3 4 5 6 7)	add 2, ..., and 7, because prefix syntax is **good**
(+ (list 2 3 4 5 6 7))	error. Even though + can "do" a sequence of numbers, it can't deal with lists.

;; *sum :* (*Listof Number*) → *Number*
;; add up the numbers in a list
(**define** (*sum l*)
  (apply + *l*))
 	
;; tests
(= (*sum* (list 1 2 3)) 6)
	With apply, we can do it anyway. Say we "launch" the *sum* application on (list 1 2 3),
	  (*sum* (list 1 2 3))
	= (apply + (list 1 2 3))
	= (+ 1 2 3)
	So apply does "the right thing."

;; *average :* (*Listof Number*) → *Number*
;; the average for a non-empty list,
;; error otherwise
(**define** (*average l*)
  (**cond**
  [(empty? *l*) (error ...)]
  [**else** (/ (apply + *l*) (length *l*))]))
	Yes, this is the two-line expert version of average.

```
;; largest :
;;       (NEListof Number) → Number
;; the largest item on the list
(define (largest l) (apply max l))
```

Yes, it's one line long.

```
;; sentence :
;;    (Listof String) String → String
;; separate the words
;;    with a leading space,
;; add a "." at the end
(define (sentence l end)
  (string-append
    (apply string-append
           (map (lambda (word)
                  (string-append
                    " " word))
                l))
    end))

;; tests:
(string=?
  (sentence
    (list "hello" "world") "!")
  " hello world!")
(string=?
  (sentence
    (list "how" "are" "you") "?")
  " how are you?")
```

Imagine doing this without apply and **lambda**.

But, always remember, you don't really ever need loops.

You can always design a function instead of using a loop.

7.3 Example: Keeping Track of Grades

Data definitions:

```
;; A gradebook (gb) is: (Listof GBE).

(define-struct gbe (name grades))
;; A GBE (grade-book entry) is:
;;       (make-gbe symbol (Listof Number))

;; Example:
(define gradebook
  (list (make-gbe 'ALICE (list 49 81 64))
        (make-gbe 'BOB   (list 16 31 27))
        (make-gbe 'CARL  (list 27 64 8))))
```

Exercise

Exercise 7.1 Design

```
;; students-in : GB → (Listof Symbol)
```

which enumerates the students in the course (duplicates are okay). ∎

Exercise 7.2 Design

```
;; grades-of : GB Symbol → (Listof Number)
```

which extracts the grades for a given student from the given *GB*. ∎

Exercise 7.3 Design

```
;; total-of : GB Symbol → Number
```

computes the total score for a given student from the given *GB*. ∎

Exercise 7.4 Design

```
;; highest-total : GB → Number
```

which determines who has the highest total score in the class. ∎

Exercise 7.5 Design

```
;; chart-grades : GB → Histogram
```

which consumes a gradebook and generates a histogram.

```
;; A Histogram is: (Listof HistoLine).

(define-struct histoline (name result))
;; A HistoLine is: (make-histoline Symbol String)
;; Note: the result string is at most 60 characters long.
```

Each item of the generated list is a name combined with a string representing the histogram entry. The longest string, representing the person with the highest grade, should be 60 characters long. Every other string should be proportionally shorter. You may need some math functions to help you compute the lengths of strings; once you've determined what functions you need, ask the TAs for help finding these in Scheme or experiment or use the HelpDesk. ∎

7.4 Input and Output: S-expressions

read *: IPort → S-expression*	Given an *IPort*, read those characters from the port that form an S-expression and produce it.
If the file is looks like this 10	the S-expression is 10.
If the file looks like this *hello*	the S-expression is 'hello.
If the file looks like this (*hello world*)	the result is (list 'hello 'world).
If the file looks like this ((*ALICE* 1) (*BOB* 2) (*CARL* 3))	the S-expression is (list (list 'ALICE 1) (list 'BOB 2) (list 'CARL 3))
In short, read adds list and **quote** (') where needed.	But where do you get an input port from?
Like this: open-input-file *: String → IPort*}	Find a file by name (string), open it, and make it available as an (input) *Port*.
So composing the two functions is useful: (read (open-input-file "gradebook.txt"))	If the name of your gradebook file is "gradebook.txt", then this reads an entire grade book.

If `"gradebook.txt"` contains this:

```
((ALICE 49 81 64)
 (BOB   16 31 27)
 (CARL  27 64 8))
```

. . . then the expression produces:

```
(list (list 'ALICE 49 81 64)
      (list 'BOB   16 31 27)
      (list 'CARL  27 64 8))
```

Of course, this isn't quite what our data definition says.

```
(define gradebook
  (map (lambda (raw-gbe)
         (make-gbe
           (first raw-gbe)
           (rest raw-gbe)))
       (read
         (open-input-file
           "gradebook.txt"))))
```

Exercise Download the data file

```
hamlet.txt
```

Study it to ascertain its structure. (Does it remind you of a gradebook?)

Exercise 7.6 Develop a data definition for representing a dialog with lists and structures. Here are some guidelines:

Add the teachpack
`io.ss`

- The entire list is a *dialog*.
- Each element of a dialog is a *part*. A part consists of a symbol, indicating which character is speaking, and a list of strings.
- Each string in the list of strings is called a *line*.

For instance,

```
((HAMLET
   "My excellent good friends! How dost thou,"
   "Guildenstern? Ah, Rosencrantz!
     Good lads, how do ye both?")
 (ROSENCRANTZ
   "As the indifferent children of the earth.")
 (GUILDENSTERN
   "Happy, in that we are not over-happy;"
   "On fortune's cap we are not the very button.")
 (HAMLET
   "Nor the soles of her shoe?"))
```

is a dialog with four parts. *HAMLET* speaks two parts, while his comrades speak one each. The first and third part each have two lines, while the second and fourth have one line each. In all, *HAMLET* speaks three lines spread over two parts in this dialog.

Note: in keeping with tradition, the names of the characters are in UPPERCASE. Thus, you must refer to the Prince of Denmark as 'HAMLET, not as 'Hamlet or 'hamlet.

Use read and open-input-file to read the file from disk. Also traverse the resulting S-expression to create instances of *Dialog* so that the remaining functions can process the dialog easily.

Hint: See how we read the gradebook and converted into a list of structures. ∎

Exercise 7.7 Use *your* data definitions from exercise 7.6 to design

;; *speakers : Dialog* → (*Listof Symbol*)

which computes the names of the speakers who contribute to a dialog. Each speaker's name must appear exactly once (though the order does not matter).

Hint: Design the helper function

;; *eliminate-duplicates :* (*Listof Symbol*) → (*Listof Symbol*)

which consumes a list of symbols and produces one without duplicates. More precisely, a symbol should only be on the result list if it is *not* in the rest of the list. ∎

Exercise 7.8 Use *your* data definitions from exercise 7.6 to design

;; *lines-of : Symbol Dialog* → (*Listof String*)

which generates a single list of all the lines spoken by the character named by the first argument. ∎

Exercise 7.9 Use *your* data definitions from exercise 7.6 to design

;; *most-parts : Dialog* → *Symbol*

which computes which speaker has the most parts in the dialog. ∎

Exercise 7.10 Use *your* data definitions from exercise 7.6 to design

;; *most-lines : Dialog* → *Symbol*

which computes which speaker has the most lines in the dialog. ∎

Exercise 7.11 Use *your* data definitions from exercise 7.6 to design

;; *abbreviate-for : Symbol Dialog* → *Dialog*

which extracts a given actor's part from a dialog. ∎

Rationale: When rehearsing for a play, it's useful to have a list of just your parts, with only an indication of when someone else is going to speak, not what they say. For the actor playing *HAMLET*, for instance, the function would yield

```
((HAMLET
  "My excellent good friends! How dost thou,"
  "Guildenstern? Ah, Rosencrantz!
     Good lads, how do ye both?")
 (ROSENCRANTZ)
 (GUILDENSTERN)
 (HAMLET
  "Nor the soles of her shoe?"))
```

for the example in exercise 7.6. ∎

8 Thursday Afternoon: A Networking Application

Whose Line is It, Anyway?
By Shriram

Say, you are part of a group of Shakespeare fans who "read" the Bard's plays over the Internet. You like reading Shakespeare (who doesn't?), but you hate typing in all those "quothe"s and "thine"s and what not (who doesn't?). Being a Seasoned Schemer, you're too smart to get suckered into typing all that text. Instead, you want to write an Internet agent that does the work for you. Let's assume there are only three roles in the dialogs we want to read: yours and two others.

Exercise

Exercise 8.1 To wit, you will write a procedure with the following contract:

;; *serve : Symbol Number Dialog String Number String Number* → true

The arguments to *serve* have the following meaning:

- The first argument, a symbol, specifies which of the roles *serve* will play.
- The second argument, a port number, specifies on which port *serve* will receive messages.
- The third argument is the dialog, which *serve* reads from the file that you created in exercise 7.6.
- The fourth and fifth arguments designate the port number and machine name for an agent playing one of the other two characters. Which character this other agent plays doesn't matter.
- The sixth and seventh arguments designate the port number and machine name for an agent playing the remaining character.

The supplementary teachpack provides two functions:

Add the teachpack `hamsup.scm`

```
;; install-listener : Number (Sexpr → true) → true
;; installs a procedure that is invoked each time when a
;; message shows up on a specific port number
(define (install-listener listener) ...)
```

```
;; send-message : String Number Sexpr → true
;; send a message to a machine at a certain port
(define (send-message machine port msg) ...)
```

Here are some more specifics on *install-listener*. The procedure given to *install-listener* receives the content of the message, which can be any S-expression. To standardize, we will use the following packet format:

```
(list symbol
      (list string)
      number)
```

The first component is the name of a character; the second is a list of strings indicating the lines in a part for that character; and the third is a counter, which we will discuss below. When your agent receives a message, it should print the name and dialog that your Internet buddies sent.

Hint: The functions display and , available in the *io* teachpack, which we installed for exercise 7.6. The first writes an S-expression to a port and produces true; the second writes a new line to a port and produces true, too. A TA will help you put these together.

The number dictates which part of the dialog (assumed to be shared by all three agents) comes *next*—indices begin at 0. Therefore, your agent needs to look up the dialog to determine who is supposed to say that part. If it's someone else, do nothing (the agent can just return a dummy value, like 'do-nothing). Otherwise, your agent needs to transmit the Bard's eternal words. It does this by using the library procedure

send-message : machine-name port-number s-expression → <ignore>

It needs to send a message to each of the other two machine/port combinations that were given as arguments to *serve*. The s-expression in this case is a packet in the format specified above. **Important**: The number your agent specifies in the packet it transmits must be one bigger than the number it received!

This process works well once initiated, but it's impossible to initiate! (A message only gets sent in response to an incoming one, so if you never receive a message....) Therefore, we shall designate one kind of packet as special: ones of the form

```
(fake () 0)
```

If the number field is 0, that means the message is being sent only to initiate the dialog. Ideally, you should not bother printing out the character's name

and dialog in this case (since they are nonsensical values), but don't worry about this initially.

As an example, a dialog using the Hamlet prose might look like this:
machineG: running

```
(serve
 'GUILDENSTERN 7000 dialog "machineR" 7001 "machineH" 7002)
```

machineR: running

```
(serve
 'ROSENCRANTZ 7001 dialog "machineG" 7000 "machineH" 7002)
```

machineH: running

```
(serve 'HAMLET 7002 dialog "machineR" 7001 "machineG" 7000)
```

Someone executes

```
(send-message "machineG" 7000 '(fake () 0))
```

(sending the same message to the other two should have no effect).

On receiving this, machineG transmits

```
(list 'GUILDENSTERN '("My honoured lord!") 1)
```

to the other two; machineR then transmits

```
(list 'ROSENCRANTZ '("My most dear lord!") 2)
```

to the other two; machineH then transmits

```
(list 'HAMLET
      '("My excellent good friends! How dost thou,"
        "Guildenstern? Ah, Rosencrantz! Good lads, how do ye both?")
      3)
```

to the other two; machineG then transmits

```
(list 'ROSENCRANTZ
      '("As the indifferent children of the earth.")
      4)
```

to the other two; and so forth. ∎

Notes on Machines, Ports and Debugging

- Your own machine always has the name "localhost". You can actually run more than one agent on your own machine, just by using different port numbers (therefore, in the example above, you could replace "machineH", "machineG" and "machineR" with "localhost").

- Port numbers below 1024 are reserved for those with administrative privilege. In general, the lower the port number, the more programs likely to be vying for it. Choose large numbers below 65536.

- Don't try to write your agent all in one go! First write a very simple listener that receives any kind of packet at all, and prints a simple message. Use *send-message* to test it. Make sure you get a feel for how the two library procedures work. Add functionality little-by-little.

- Eventually, we'd like you to "play Hamlet" with two other people in the class. To test, however, you will want to play all three roles yourself. Once you've written your version of *serve*, you might be inclined to run it three times from the Interactions window of DrScheme. If the main body of *serve* invokes *install-listener*, you're in luck: *install-listener* creates a new *thread*, so when you run *serve* you ought to see a response like this:

```
> (serve
    'HAMLET 7002 dialog "machineR" 7001 "machineG" 7000)
true
>
```

  with a fresh prompt awaiting your input.

9 Friday Morning: Games

See 'net, HtDP+

9.1 Goal

to understand the world of reactive programming in the **draw.ss** teachpack

9.2 Calling Functions on Events

World	A *World* is a collection of things, which can also be displayed on a canvas. Usually we represent a world as struct.
Show and tell: (*start Number Number*)	Display the canvas for the world.
(*big-bang Number World*)	The function *big-bang* "creates" the world. The number tells it how fast to tick the real clock; the given world is the first world. Every tick of the clock can update the world.
tick, (*on-tick-event tock*)	DrScheme calls the function *tock* for every tick of the clock.
click, (*on-key-event react*)	DrScheme calls the function *react* for every key stroke on the keyboard.
(**draw** (*draw-solid-rect* (make-posn 0 0) 100 100 'blue) **produce** 1)	draw something, then return a value (note: **produce** is a new keyword)

9.3 Example: Flying a Rocket

Rockets:

```
(define RKT-WD 3)
(define RKT-HT 8)
(define RKT-X 16)
(define FLM 20)

;; N → Number
;; the height of the rocket at t
(define (rocket-y t) (- HEIGHT (* 1/2 G (sqr t))))

;; N → true
;; draw a rocket
(define (rocket-draw t)
  (local ((define y (rocket-y t)))
    (and
     (draw-solid-rect (make-posn RKT-X y) RKT-WD RKT-HT 'blue)
     (draw-solid-rect
       (make-posn RKT-X (+ y RKT-HT)) RKT-WD FLM 'red))))
```

World is: a natural number (time in ticks)

```
(define G 3)
(define WIDTH 100)
(define HEIGHT 300)

;; World → true
(define (world-draw w) (rocket-draw w))

;; World → true
(define (world-clear w)
  (draw-solid-rect (make-posn 0 0) WIDTH HEIGHT 'white))

;; World → World
;; erase old world, draw new one, return new one
(define (tock w)
  (local ((define new-w (+ w 1)))
    (draw (world-clear w) (world-draw w) produce new-w)))
```

Run, program run!

```
(start WIDTH HEIGHT) ;; show the canvas
(big-bang .2 0) ;; create the world, start the clock
(on-tick-event tock)
```

9.4 The Nature of Keystroke Events

Data definition:

```
;; The KeyEvent is one of:
;; — Character
;; — Symbol
;; Interpretation: a character denotes an ordinary alpha-numeric key
;; a symbol denotes arrow keys ('left, 'right, 'up, 'down) or other events.
```

Data examples: #\a, #\−, 'up
Template:

```
;; KeyEvent → ???
(define (process-event ke)
  (cond
    [(char? ke) ...]
    [(symbol? ke) ...]))
```

Example:

```
(define-struct kc (l r u d))
;; Kc = (make-kc N N N N)

;; KeyEvent Kc → Kc
;; add 1 to the appropriate part of ct
(define (arrow-key-counting ke ct)
  (cond
    [(char? ke) ct]
    [(symbol? ke)
     (cond
       [(symbol=? 'left ke)
        (make-kc (+ (kc-l ct) 1) (kc-r ct) (kc-u ct) (kc-d ct))]
       [(symbol=? 'right ke)
        (make-kc (kc-l ct) (+ (kc-r ct) 1) (kc-u ct) (kc-d ct))]
       [(symbol=? 'down ke)
        (make-kc (kc-l ct) (kc-r ct) (+ (kc-u ct) 1) (kc-d ct))]
       [(symbol=? 'up ke)
        (make-kc (kc-l ct) (kc-r ct) (kc-u ct) (+ (kc-d ct) 1))]
       [else ct])]))

;; test:
(equal? (arrow-key-counting 'left (make-kc 1 0 2 9))
        (make-kc 2 0 2 9))
```

9.5 Example: Launching a Rocket

Say we want to launch the rocket with a keystroke on 'up

```
;; The world is one of:
;; — false
;; — N
;; Interpretation: false says no rocket has been launched; a
;; natural number says the rocket has been flying for that many ticks

(define G 3)
(define WIDTH 100)
(define HEIGHT 300)

;; World → true
(define (world-draw w) (rocket-draw w))

;; World → true
(define (world-clear w)
  (draw-solid-rect (make-posn 0 0) WIDTH HEIGHT 'white))

;; World → World
;; erase old world, draw new one, return new one
(define (tock w)
  (cond
    [(boolean? w) w]
    [else (local ((define new-w (+ w 1)))
            (draw (world-clear w) (world-draw w) produce new-w))]))

;; KeyEvent World → World
;; change the world to a rocket
(define (react ke w)
  (cond
    [(char? ke) w]
    ;; now we know it's a symbol
    [(symbol=? 'up ke) 0]
    [else w]))

;; — run program run
(start WIDTH HEIGHT) ;; show the canvas
(big-bang .2 false) ;; create the world, start the clock
(on-tick-event tock)
(on-key-event react)
```

Exercise

Star Money, Star Thalers by the Grimm Brothers

> There was once upon a time a little girl whose father and mother were dead, and she was so poor that she no longer had a room to live in, or bed to sleep in, and at last she had nothing else but the clothes she was wearing and a little bit of bread in her hand which some charitable soul had given her. She was good and pious, however. And as she was thus forsaken by all the world, she went forth into the open country, trusting in the good God.
>
> Then a poor man met her, who said, "Ah, give me something to eat, I am so hungry."
>
> She handed him the whole of her piece of bread, and said, "May God bless you," and went onwards.
>
> Then came a child who moaned and said, "My head is so cold, give me something to cover it with."
>
> So she took off her hood and gave it to him. And when she had walked a little farther, she met another child who had no jacket and was frozen with cold. Then she gave it her own, and a little farther on one begged for a frock, and she gave away that also.
>
> At length she got into a forest and it had already become dark, and there came yet another child, and asked for a shirt, and the good little girl thought to herself, "It is a dark night and no one sees you, you can very well give your shirt away," and took it off, and gave away that also.
>
> And as she so stood, and had not one single thing left, suddenly some stars from heaven fell down, and they were nothing else but hard smooth pieces of money, and although she had just given her shirt away, she had a new one which was of the very finest linen. Then she put the money into it, and was rich all the days of her life.

Exercise 9.1 Develop a game program based on the story of "Star Money, Star Thalers." The *main* function of the program should consume a natural number and drop that many thalers (from the top of the world) on the girl (at the bottom of the world), *one at a time*. The thaler should move randomly to the left or right and downwards, but should always stay within the boundaries of the world (canvas). The girl should react to 'left and 'right keystrokes, moving a moderate number of pixels in reaction but always staying completely within the boundaries of the world.

Add the teachpack
`draw.ss`

Hints and suggestions:

1. Represent the girl visually as a container that is open at the top (the outline of a rectangle without top boundary). Show the number of thalers it has caught inside of the container.

Design a function for drawing the girl.

2. Represent the thaler visually as a yellow disk.

   Design a function for drawing the thaler.

3. The world of StarThalers contains two physical objects: the girl and one flying thaler. It also contains two abstract objects: the number of thalers yet to come and the number of thalers that the girl caught so far.

   Pick physical dimensions of the world (width, height, background color).

   Design a function for drawing the world, assuming that the world's canvas is visible.

4. Define the function *tock*, which refreshes the visible world in reaction to time events:

   (*start WORLD-WIDTH WORLD-HEIGHT*)
   (*big-bang* .1 . . . *some initial world* . . .)
   (*on-tick-event tock*)

5. Design a function for moving the world and another one for moving a thaler. Then modify *tock* so it too can move the world.

6. Design a function for moving the girl in reaction to a keystroke. The function consumes a girl and a keystroke. Its result is the girl in the new position.

   Define the function *react*, which uses this new function to move the girl in the world:

   (*start WORLD-WIDTH WORLD-HEIGHT*)
   (*big-bang* .1 . . . *some initial world* . . .)
   (*on-tick-event tock*)
   (*on-key-event react*)

7. Make sure that the world reacts (at tick events) to the landing of a thaler, to the girl catching a thaler, and to the world running out of thalers in addition to moving the thaler as needed.

Good luck and enjoy! ∎

10 Friday Afternoon: More on Games

10.1 Goal

to modify an existing program and to understand how iterative refinement and the design recipe help with this task

Exercises

Exercise 10.1 Modify the StarMoney game so that all game thalers are in the air at any point during the game.

Modify the StarMoney game so that many but not all game thalers are in the air at any point during the game. Add thalers as they land and/or are caught. ∎

Exercise 10.2 Modify the function that moves the girl so that it can fly in response to all four arrows (↑, ↓, ←, →).

Exercise 10.3 Implement some other game. ∎