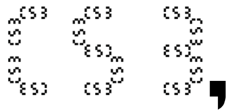
University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2008 Instructor: Dan Garcia 2008-12-18

, The Final Exam

Last name______________________________ First name ____________________________

SID Number____________________________ TA's name ____________________________

(Sorry to ask this next question, but with 100+ students, there may be a wide range of behavior.)

The name of the student to my left is ____

The name of the student to my right is ____

I certify that my answers are all my own work. I certify that I shall not discuss the exam with anyone in CS3 who has yet to take it until after the exam time.

Signature ____

Instructions

- Question 0: Fill in this front page and write your name on the front of every page! The exam is open book and open notes (no computers). Put all answers on these pages; don't hand in any stray pieces of paper.
- **You may NOT write any auxiliary functions for a problem unless they are specifically allowed in the question**.
- Feel free to use any Scheme function that was described in sections of the textbook we have read without defining it yourself.
- You do not need to write comments for functions you write unless you think the grader will not understand what you are trying to do otherwise.
- You have three hours, so relax. We estimate 30 minutes per question.
- Feel free to write λ instead of `lambda`.
- Feel free to write any comments you wish in the margins of this front page. Good skill!

Grading Results

Question	*Max. Points*	*Points Given*
0	**0 / -1**	
1	**15**	
2	**20**	
3	**20**	
4	**20**	
5	**10**	
6	**15**	
Total	**100**	

Login: `cs3-___`

Question 1 – This Blankety blank blank exam… (15 pts, 1 pts each; 30 min)

Fill in the blanks below. The symbol "➔" means "evaluates to".

- If any of the following display an *error*, write "ERROR" & *describe the error*.
- If any of the following go into an *infinite loop*, write "INFINITE LOOP".
- If any of the following are *impossible*, write "IMPOSSIBLE".
- If any of the return values are *procedures*, write <PROCEDURE *name*> (e.g., `cons` ➔ `<PROCEDURE cons>`)

```
(define (mystery f g) (f (g f)))
(define (call f n)    (f n))
(define (1+ n)        (+ n 1))
(define (double n)    (* n 2))
(define (square n)    (* n n))
```

```
(define (aced-cs3-final? score)
  (if (equal? score (or 'seventy-five 75))
      'yes
      'no))
```

a) `(bf (bl (word 1 (word 2 (word 3 "" )))))` ➔ ______

b) `(bf (bl (se 1 (se 2 (se 3 '() )))))` ➔ ______

c) `(car (cdr (cons 1 (cons 2 (cons 3 '() )))))` ➔ ______

d) `(car (cdr (list 1 (list 2 (list 3 '() )))))` ➔ ______

e) `(car (cdr (append '(1) (append '(2) (append '(3) '() )))))` ➔ ______

f) `(bf (first ______________________ ))` ➔ `(cal)`

g) `(cdr (car ______________________ ))` ➔ `(cal)`

h) `(aced-cs3-final? 75)` ➔ ______

i) `(if (aced-cs3-final? ______________ ) 'a+ 'f)` ➔ `f`

j) `(mystery ______________ ______________ )` ➔ `(cal)`
No explicit λ in this blank

k) `(reduce call (list ______________________ 3))` ➔ `65`
You may only use **`1+`**,**`double`**, *or* **`square`** *but you may use up to 4 of them here*

l) A practical application for the *Dragon Curve* fractal would be…
(Hint: How would Dan use them if he were building a new house?)

____.

m) If you're playing misére "1, 2, …, 10" against Gamesman, you want to play…

FIRST / SECOND / DOESN'T-MATTER-GAMESMAN-ALWAYS-WINS (circle one).

n) We want to write a `num-arguments` to return how many arguments a function call has:

```
(num-arguments (list 'a 'b)) ➔ 2
(num-arguments (+ 10 21 40)) ➔ 3
(num-arguments (no-arg-fun)) ➔ 0
```

Now, write `num-arguments`:

`(define (num-arguments expression) ______________________ )`

Login: `cs3-___`

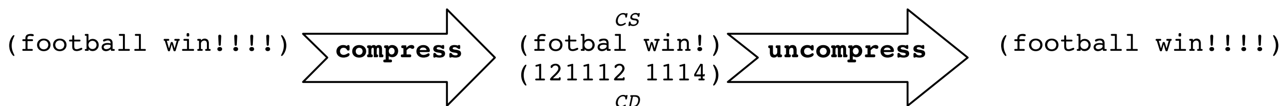
Question 2 – Cal beat HOFstra with a full-court uncompress... (20 pts; 30 min.)

You've probably been annoyed by the fact that `every` only takes a *unary* function and a *single* word/sentence. What if we had a cool *binary* function `f` and *two* words/sentences and wanted to call `f` on every *pair of letters or words* in them? We'd be stuck! Well, here is `every2`, which exactly does what we want!

```
(define (every2 f SW1 SW2)  ;; f is a binary function; SW1 & SW2 are sents/words
  (if (or (empty? SW1) (empty? SW2))
      '()
      (se (f (first SW1) (first SW2))
          (every2 f (bf SW1) (bf SW2)))))

(every2 word '(gol b) '(den ears)) ➔ (golden bears)
(every2 word  'gol     'den)       ➔ (gd oe ln)
```

That was the setup; here's the cool problem. *Compression* is the idea that you can save space by shrinking something down, transferring it, and then unshrinking it later. We decide to do the same thing with *sentences* to eliminate repeat letters. We'll compress sentences by creating two sentences, which we'll call CS and CD. CS has the repeated letters (up to 9) removed, and CD tells us (for each letter in each word in CS) how many times to repeat. All we care about is *uncompression*; assume *compression* is done for us.



Your job is to write **`uncompress`**. You may write *any helper functions you wish*, but **YOU MAY NOT USE EXPLICIT RECURSION**; instead, use HOFs.

```
;; (uncompress CS CD)
;; INPUTS      : CS (A compressed sentence. C stands for "Compressed")
;;             : CD (An identical-length sentence with each Digit indicating how
;;             :     many times the corresponding letter in CS should be repeated.)
;; RETURNS     : The original sentence before compression
;; EXAMPLE     :
;; (uncompress '(woho goal! i win 10!)
;;             '(1213 19111 1 111 142)) ➔ (woohooo gooooooooal! i win 10000!!)
```

```
(define (uncompress CS CD) ;; Helpers might make this problem easier…
                           ;; Don't use any EXPLICIT RECURSION;instead use HOFs
                           ;; (even the helpers may not use explicit recursion)
```

Login: `cs3-___`

Question 3 – Run past the third car and then go deep… (20 pts; 30 min.)

You have probably found `list-ref` very useful. We'd like to write `deep-list-ref` which takes a deep list `L` and an index `i` and finds the `i`th atom (starting at 0) in `L`:

```
(deep-list-ref '(i ((love) cs3) exams) 0) ➔ i
(deep-list-ref '(i ((love) cs3) exams) 1) ➔ love
(deep-list-ref '(i ((love) cs3) exams) 2) ➔ cs3
(deep-list-ref '(i ((love) cs3) exams) 3) ➔ exams
(deep-list-ref '(i ((love) cs3) exams) 4) ➔ ERROR: car: wrong type of argument
```

```
(define (deep-list-ref L i)
  (dlr-helper L i 0))

  (define (dlr-helper L i seen)
   (cond
1        ((list? (car L))
2         (dlr-helper (cons (car L) (cdr L)) i seen))
3        ((= i seen)
4         (car L))
5        (else (dlr-helper (cdr L) i (+ seen 1)))))
```

a) There is only one buggy line above. The bug affects some (but not all) input. Fill in the blanks below to complete the sentence. All arguments to `deep-list-ref` should be *as short & simple as possible*.

"Calling `(deep-list-ref` ______________________________ `1 )`

doesn't trigger the bug and correctly returns ______________________."

b) "However, calling `(deep-list-ref` ______________________________ `1 )`

causes/returns (fill in description) ______________________________

when it should return ______________________________."

c) "Replacing line ____ with ______________________________
fixes the bug so that `deep-list-ref` works as advertised on all input."

d) **After you make the change** in (c) your friend asks you an interesting question. "What if you changed the order of the first two `cond` statements?" I.e., you swap lines (1 & 2) with lines (3 & 4). What would be the result of the following calls?

```
(deep-list-ref '(i ((love) cs3) exams) 0) ➔ ____________ ;; want i

(deep-list-ref '(i ((love) cs3) exams) 1) ➔ ____________ ;; want love

(deep-list-ref '(i ((love) cs3) exams) 2) ➔ ____________ ;; want cs3
```

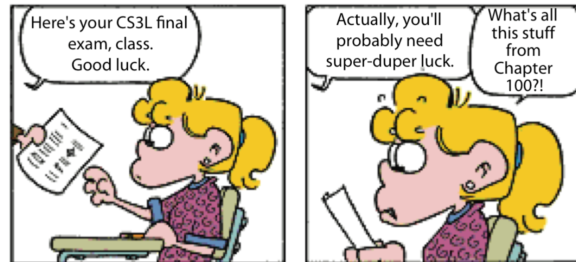
Put down your pen or pencil, stretch, take a deep breath, and proceed

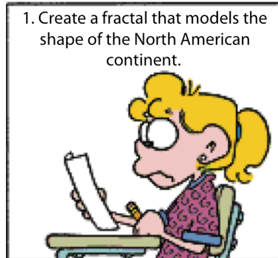
If you followed our suggested pace, you should have 90 min left and be half done.

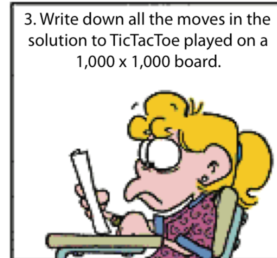
FoxTrot

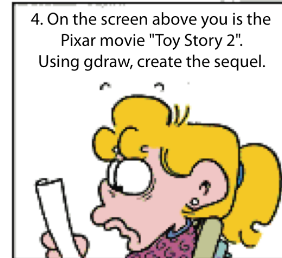
BILL AMEND

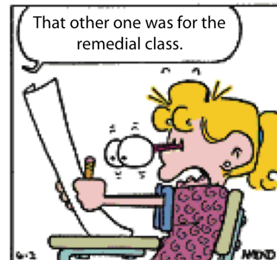
(with CS3L customizations)











Login: `cs3-___`

Question 4 – `fastest-game` in the west… (20 pts; 30 min.)

Have you ever played a board game with a friend and wanted to see how fast the game could theoretically end? That is, if the goal for both players were to finish the game as fast as possible (regardless of who won), how many moves would be made? As a reference, here are the brief specs for the *big-3* (most important functions):

- `(primitive-position pos)` ➔ `w, l` or `t` if `pos` is *primitive* (end of game), else `#f`
- `(do-move pos move)` ➔ A *new* position, the result of making that `move` at that `pos`
- `(generate-moves pos)` ➔ A list of *all the moves* available at position `pos`

For this problem, we will deal only with *non-loopy* games (i.e., games which never have repeat positions that could cause the game to end in a draw). This is like exactly like the games you've seen: *1,2,…,10*, *Tomorrow's Tic-Tac-Toe*, *Northcott's game* (except pieces always move towards the middle), *Knight's Dance* (except you can't move to a square you've already occupied), *Konane,* and *Surround*.

a) Write `next-positions` that takes a *position* and returns a *list* of all of the positions which can be reached from that position in exactly one move. **Your solution should work for any game** in which the *big-3* (above) are defined. (7 pts)

```
;; Example for "1,2,…,10" in which a position is represented as (L/R num)
;; & a move is either "1" or "2". From 0, you can reach 1 or 2.
(next-positions '(L 0)) ➔ ((R 1) (R 2))

(define (next-positions pos)

    ________________________________________________________________________ )
```

b) Write `fastest-game` which should return the number of moves in the fastest possible game. **It should work for ANY non-loopy game!!** You <u>**may not**</u> write any helper functions; just fill in the blanks below. (13 points)

```
;; INPUTS      : pos (The representation of a position of the game)
;; REQUIRES    : (1) primitive-position, do-move and generate-moves
;;             :     already defined for the particular game.
;;             : (2) pos must be a valid position
;;             : (3) The game cannot be loopy
;; RETURNS     : The number of moves in the fastest possible game (possibly 0)
;; EXAMPLES    : Assuming primitive-position, do-move & generate-moves for
;;             : the game "1,2,…,10", which encodes its position as (L/R num)…
;;             : (fastest-game '(L 0)) ➔ 5   E.g., 0→2→4→6→8→10 (5 moves)
;;             : NOTE: The representation of positions differs with each game.
;;             :       Assume fastest-game will be called with a valid position
;;             :       for that particular game (don't do any error-checking).

(define (fastest-game pos)

  (if ___________________________________

      ___________________________________

      ________________________________________________________________________ ))
```

Login: `cs3-___`

Question 5 – Magical Mystery Fractal, step right this way… (10 points; 30 min)

We've seen two remarkable fractals, the *C-Curve* and the *Dragon Curve*. They are shown below for the first three generations, 0 through 2. The `U` label helps us remember which direction "up" is – the curve will "bulge" towards it.

a) Draw the n= 2, 3, 4 and ∞ generations below for the new *Mystery Curve*. You don't have to draw the arrowheads – we just draw them to highlight the differences between the fractals. If you *do* draw the arrowheads, draw them very small. If you're stuck, it might help to draw another iteration of *C-Curve* or *Dragon Curve*.

	n=0	n=1	n=2	n=3	n=4	…	n=∞
C-Curve	U	U U		…			
Dragon Curve	U	U U		…			
Mystery Curve	U	U U				…	

(Use the area below for scratch; put your answers in the boxes above when you're done.)

a) Below is the code for the *Dragon Curve* that you've seen before. Change as few lines as possible so that it implements the *Mystery Curve*. When changing a line, circle the number above and write the new line to the right of it.

```
(define (fractal P0 P1 n)                          ;; 1
  (if (= level 0)                                  ;; 2
      (draw-line-P P0 P1)                          ;; 3
      (let ((Pm (coord-P P0 P1 (make-V -½ ½))))    ;; 4
         (fractal P0 Pm (- n 1))                   ;; 5
         (fractal P1 Pm (- n 1)) )))               ;; 6
```

Pm

P0 P1

Login: `cs3-___`

<u>Question 6 – It's a small world after all (aka MapReduce) (15 points; 30 min)</u>

For both problems, assume `identity` is defined as usual: `(define (identity n) n)`

a) Fill in the blanks below. The file `"/randoms"` has 10 random numbers, and `"/shakesp"` has the collected works of William Shakespeare.

```
(reduce-map-_______ ____________________________ ____________________________ "/randoms")
;; return largest digit in "/randoms"

(reduce-map-_______ ____________________________ ____________________________ "/randoms")
;; return #t if 42 (my lucky number) is in "/randoms", #f otherwise

(reduce-map-_______ ____________________________ ____________________________ "/shakesp")
;; return number of lines Shakespeare wrote
```

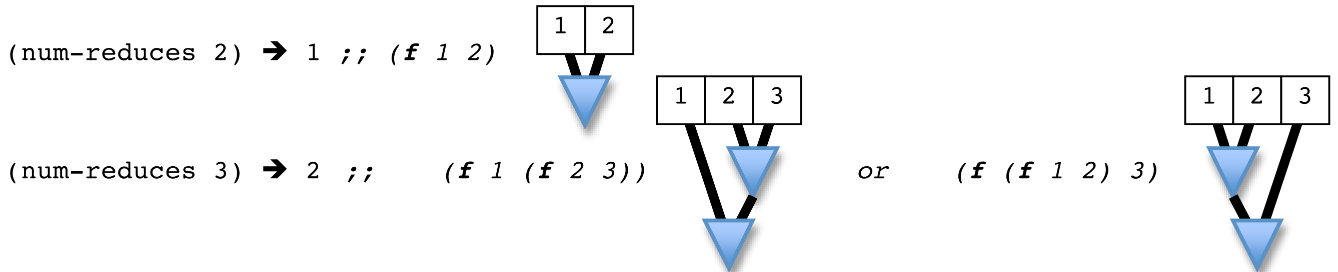
We modify `reduce-map-word` so the *left-to-right ordering of the inputs is preserved* in the map and reduce stage, but the reducing can be paired up any way it wants. There can be many equally possible reductions. How many, though? Let's see a specific example.

b) How many possible reductions (as a function of `n`) are there for a call to the function `(reduce-map-word f identity "numbers")`, if the `numbers` file contains the numbers `1` through `n` on different lines? You may find `1upto` useful, which returns a sentence of the numberss `1` *upto (not including)* `n`. E.g., `(1upto 5)` ➔ `(1 2 3 4)`

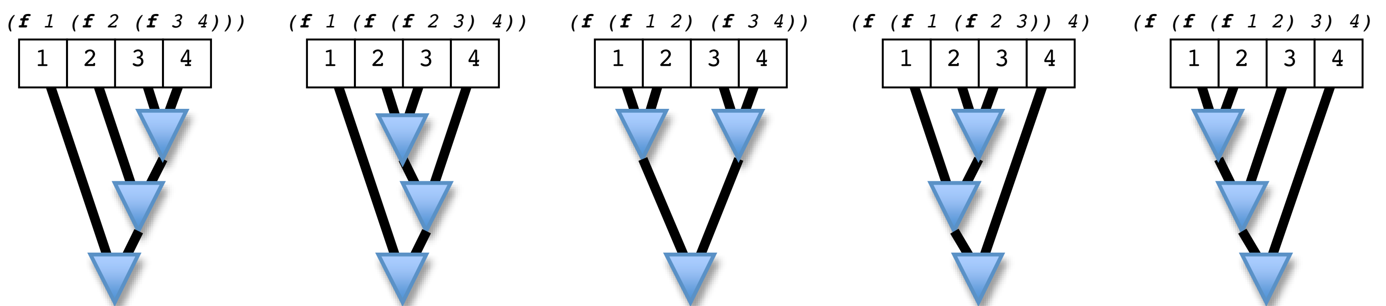
```
(num-reduces 1) ➔ 1 ;; 1 ← (reduce f '(1)) doesn't even call f, so 1 reduction

(num-reduces 2) ➔ 1 ;; (f 1 2)

(num-reduces 3) ➔ 2 ;;    (f 1 (f 2 3))          or    (f (f 1 2) 3)
```




```
(num-reduces 4) ➔ 5 ;; Below. See the pattern? E.g., (num-reduces 3) is in there…
```




```
(num-reduces 5) ➔ 14 ;; Too many to print

(define (num-reduces n)
   (if (< n 3)
       1

       ______________________________________________________________________

       ______________________________________________________________________ ))
```