University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2002 | Instructor: Dan Garcia | 2002-12-17

CS3 Final Exam

Last name______________________________ First name ______________________________

SID Number______________________________ TA's name ______________________________

(Sorry to ask this next question, but with 200+ 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.
- Each question is worth the same amount, so don't spend all your time on one problem; if you're stuck, move on.
- 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	**20**	
2	**20**	
3	**20**	
4	**20**	
5	**20**	
6	**20**	
Total	**120**	

Question 1 – This Blankety blank blank exam... (20 pts, 3 pts each; 30 min)

(If you get every question on this page correct, you earn one bonus point)

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>`)

a) `(butfirst (butlast (se '(this) 'is '(easy) )))` ➔ (is)

b) `(cdadr` '(() (a)) `)` ➔ `()`

c) `(cdadr` impossible `)` ➔ `cs3`

d)
```
(define foo (lambda (arg) arg))
(filter foo (list "" 'if '() #f not #t))
```
➔ ("" if () not #t)

e) `(equal? / (reduce` (lambda (proc1 proc2) (if (= (proc1 9 3) 3) proc1 proc2)) `(list + / * -)))` ➔ `#t`

```
(lambda (proc1 proc2)
   (if (= (proc1 9 3) 3)
       proc1
       proc2))
```

f)
```
(define (1+ n) (+ 1 n))
(define (bar n) ((repeated 1+ n) n))
(every bar '(1 2 3))
```
➔ (2 4 6)

g)
```
(define (double x) (* 2 x))  ;; Fill in nothing but parens in the parens blanks
(define mystery              ;; to complete an expression that evaluates
  (lambda (x)                ;; to the number in the must be a number blank
    (lambda (g)
      (g (g x)))))
```

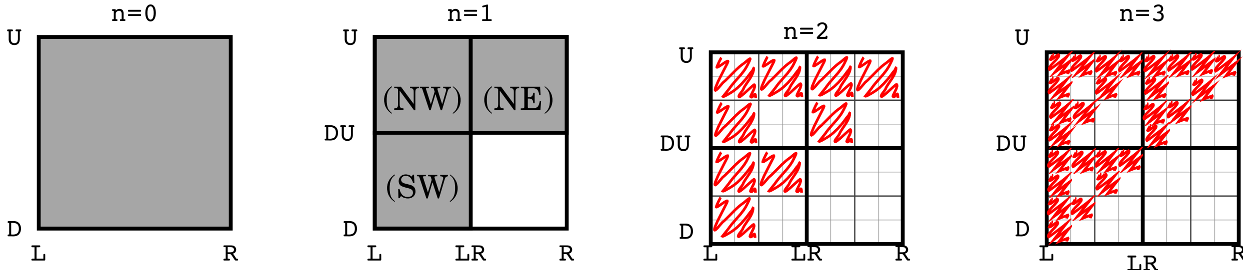
((`mystery` ((`mystery` ___ `2`) `double`)) `double`) ➔ 32

parens — *parens* — *parens* — *parens* — *parens* — *parens* — *must be a number*

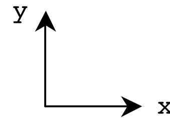
Name: ________________________________

Question 2 – Anyone up for a game of `checkers`?... (20 points; 30 min)

Let's design a fractal "checker" whose base (`n=0`) case is a filled square, and whose recursive case places three half-sized copies of the previous generation in the Northwest (NW), Northeast (NE) and Southwest (SW) corners. For example, here are the `n=0` and `n=1` fractals:



a) Sketch the `n=2` fractal above on the right. (2 points)
b) Sketch the `n=3` fractal above on the right. (3 points)
c) Fill in the blanks below to complete the definition for `checker`. (9 points)

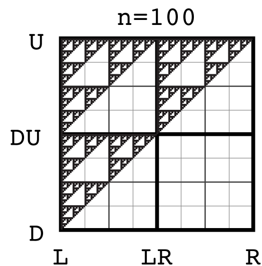



```
(define (checker L D R U n)
  (if (= n 0)
      (draw-rectangle L D R U)
      (let ((LR (/ (+ L R) 2) )
            (DU (/ (+ D U) 2) ))
          (checker L D LR DU (- n 1))
          (checker L DU LR U (- n 1))
          (checker LR DU R U (- n 1))
      )))
```

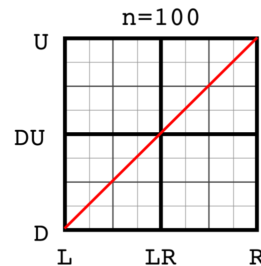
```
;; (draw-rectangle x1 y1 x2 y2)
;; LR = midway between L and R
;; DU = midway between D and U

;; you may not need every blank…
```

d) The fractal `(checker L D R U 100)` looks like a *Sierpinski triangle rotated 45°!* Let's say you forgot to write your Northwest (NW) recursive call in part (c) above. Roughly sketch what `(checker L D R U 100)` would look like below. (6 points)



A Sierpinski triangle rotated 45° is the fractal `(checker L D R U 100)` ***with*** the NW recursive call.



Sketch your answer above for the fractal `(checker L D R U 100)` ***without*** the NW recursive call.

Question* `for` *– (Get it?) (20 pts; 30 min.)

This problem concerns the definition and use of a new higher-order function called `for`. Similar to `for-each`, it allows us to repeatedly evaluate a unary procedure `f` (of `i`, say), but each time `i` increases (by some `INCR` procedure) from `FROM` up through (and *including*, if it is equal to) the bigger `TO`. When it's done incrementing (`i` > `TO`), `for` should return `done`.
We'll circle the return value for this problem to differentiate return value from display.

```
;; (for FROM TO INCR F)
;; INPUTS       : FROM (A number)
;;              : TO   (A number)
;;              : INCR (A unary procedure which takes an input number and returns
;;              :       another number hopefully closer to TO than the input)
;;              : F    (A unary procedure which gets invoked upon every iteration
;;              :       as we use INCR to increase the number until > than TO)
;; RETURNS      : [done]
;; EXAMPLES     :
;; (define (1+ x) (+ x 1)) ;; example INCR procedure
;; (define (5+ x) (+ x 5)) ;; example INCR procedure
;; (for 1 9 1+ display) → 123456789[done]
;; 123456789 were displayed, done was returned
;; (for 1 9 5+ display) → 16[done]
;; 16 were displayed, done was returned. Since 1 <= 9, (f 1) = (display 1)
;; called, then (incr 1) = (5+ 1) = 6, and since 6 <= 9, (f 6) = (display 6)
;; called, then (incr 6) = (5+ 6) = 11, and since 11 > 9 we return done.
```

a) Write `for`. **You may not define any helper functions**. (6 points)

```
(define (for FROM TO INCR F)
  (if (>= FROM TO)
      'done
      (begin (F FROM) (for (INCR FROM) TO INCR F)) ))
```

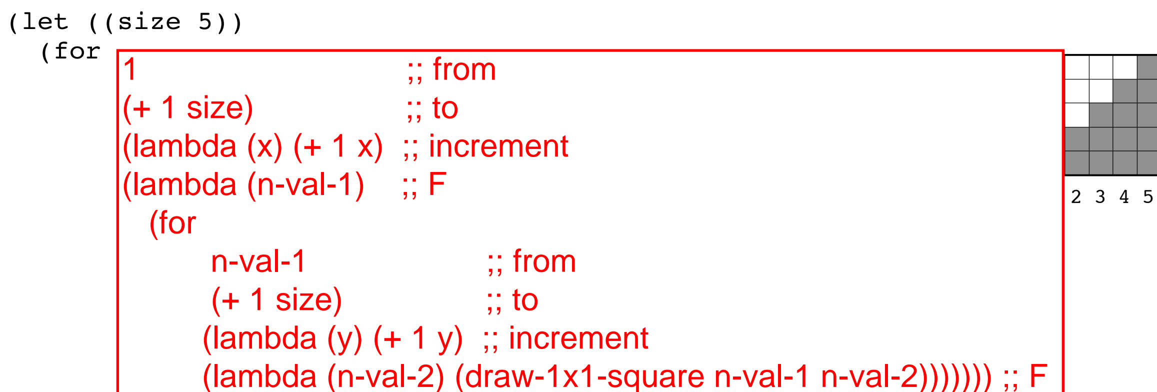
b) What's the result? Show the display & **circle** the return value (as above). (4 pts)
You should be able to do this question and part (c) below <u>*without*</u> *part (a) above.*

```
(define (square x) (* x x))
(define (silly  i)
   (display (word 'cal i))
   'is
   (+ i 1))
```

`(for 2 4 square silly)` → cal2done

c) Your are given `(draw-1x1-square x y)` which draws a 1×1 square on the screen at coordinates `(x,y)`. Use `for` to create a single expression (without defining any new helper functions) to create the 5×5 triangle shown below. Each 1×1 square should be drawn once and only once, and *by simply changing the* `5` *to* `100` *in the* `let` *your expression should draw a 100×100 triangle*. (10 points)

```
(let ((size 5))
  (for
```



Name: ______________________________

Question 5 – At Harvard, it's `caaaaaar` and `cdr`... (20 pts; 30 min.)

We all recall fondly the short-cut `c_R` functions that start with `c`, have some number of `a`'s and `d`'s and end with `R` that are used for walking down a complex list. E.g., `(cadar '((1 2) (3 4))) ==> 2` Let's assume that there is *no limit* to the number of `a`'s and `d`'s one can use between the `c` and the `R` (normally the maximum number is 4).

You are to help us debug the function `C_R` which takes an element `ELT` (not a list) and list `L` and returns the word of `a`'s and `d`'s that would have had to be used between the "`c`" and the "`R`" to extract `ELT` from the list `L`. If `ELT` is not in the list `L`, `C_R` should return `#f`.
You are told `ELT` is either in there exactly once or not at all. E.g.,

```
(C_R  2 '((1 2) (3 4)))           → ada     ;; (cADAr '((1 2) (3 4))) → 2
(C_R 'e '(a b c d e f))           → adddd   ;; (cADDDDr '(a b c d e f)) → e
(C_R 'stanford '(big game champ)) → #f      ;; stanford not in the list…
```

```
;; Assume null? has been implemented as: (define (null? arg) (equal? arg '()))
```

```
(define (C_R elt L)
  (cond ((null? L)                                ;; 1 done?
         #f)                                      ;; 2
        ((equal? elt (car L))                     ;; 3 is elt the car?
         'a)                                      ;; 4
        ((C_R elt (car L))                        ;; 5 is elt in the car?
         (word  (C_R elt (car L)) 'a))            ;; 6
        ((C_R elt (cdr L))                        ;; 7 is elt in the cdr?
         (word  'd (C_R elt (cdr L))))            ;; 8
        (else #f)))                               ;; 9 nope, then false
```

a) Fill in the blanks below to complete the sentence.
The first blank should contain the *shortest length list possible*. (10 points)

"Calling `(C_R 'x` '(a) `)` causes (fill in description of error)

'ERROR! car: wrong type of arg: a' when it should return #F.

Replacing line 1 with (cond ((or (null? L) (not (list? L)))

fixes the error."

b) **After** you make the change in (a) there is still one remaining logical bug. Fill in the blanks below to complete the sentence. The first blank should contain the *shortest length list possible*. After the fix the program should work as advertised. (10 pts)

"Calling `(C_R 'x` ____________________ `)` returns ______________________"

when it should return ____.

Replacing line ____ with ____

fixes the error."

Question 6 – A great `value`-added question… (20 pts; 30 min.)

This question deals with your project and game theoretic *values* for positions: (win=`w`, tie=`t`, lose=`l`). For now, 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) with the players alternating turns. **Do NOT use the tree abstraction for this problem**.

As a reminder:

- A ***winning*** position is one with *at least one losing child*
- A ***tieing*** position is one with *no losing children but at least one tie child*
- A ***losing*** position is one with *all winning children*

As a reference, here are the brief specs for the three 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 `pos`

Write `value` which should return the game's value as the letters `w`, `l` or `t`.
It should work for ANY non-loopy game with players alternating turns!!
You **<u>may not</u>** write any helper functions; just fill in the blanks below. (20 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
;;              : (4) Players must alternate moves
;; RETURNS      : w, l or t (depending on the computed value of the game)
;; EXAMPLES     : Assuming primitive-position, do-move & generate-moves for
;;              : the game "1,2,…,10":      (value '(1 0)) ➔ w

(define (value pos)

  (if ____________________________________________________________

      ____________________________________________________________

      (let (( child-values ___________________________________________

                           ___________________________________________ ))

         (cond (( ____________________________ ) __________ ) ;; case 1

               (( ____________________________ ) __________ ) ;; case 2

               ( else _______________________________ ))))) ;; case 3
```

YOU'RE DONE! HAPPY HOLIDAY!