University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Spring 2009 Instructor: Dan Garcia, Ph.D. 2009-03-18

CS3L Midterm

```
(define (recursion) (recursion))
```

Personal Information

Last name	**ANSWER KEY**
First Name	
Student ID Number	
Login	`cs3-`
The name of your TA (please circle)	Aditi DavidW DavidZ
Name of the person to your Left	
Name of the person to your Right	
All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS3 who have not taken it yet. ***(please sign)***	

Instructions

- Please turn off all cell phones. Remove all hats & headphones.
- You have three hours to complete this midterm. It is open book and open notes, no computers.
- Partial credit will be given for incomplete / wrong answers, so please write down as much of the solution as you can.
- **Use** `true` **instead of** `#t` **,** `false` **instead of** `#f`, since they are equivalent. Handwritten `#t` and `#f` unfortunately look too much alike…
- Feel free to write λ instead of `lambda`.
- Write the difficulty and fairness ratings in the boxes to the right and please add additional comments below.

Grading Results

Question	*Max. Pts*	*Points Earned*	*Difficulty (0=easy 5=hard)*	*Fairness (0=fair 5=unfair)*
1	15	15		
2	10	10		
3	10	10		
4	10	10		
5	15	15		
6	20	20		
Total	80			

Please comment above & left:

Login: `cs3-_____`

Question 1: This midterm comes in like a lion, goes out like a Lambda (15 pts)

a) You love recursion so much, you start doing *everything* recursively. In English, describe an embedded recursive algorithm for drinking a smoothie. *(Hint: it's similar to the algorithm we used to count the stairs in lecture.)* (6 pt)

To drink a smoothie: Ask: is the smoothie empty? If so, stop. Otherwise, take a sip and recurse (i.e., follow these directions to drink a smoothie).

b) What are *two* reasons we might use a `let`? (4 pts)

(1,2) To remove redundant code for style/performance reasons, (3) to remember a random #, (4) To make our code easier to read by giving a name to an expr

c) Add **only** parens and quotes so the expression returns an empty sentence. (5 pt)

```
((lambda ()                    ((lambda () '())) ))
(((lambda ()                    (lambda   () '()) )))
(((lambda () '()                (lambda () '()))))
 ((lambda                       (lambda) '()) '())
 ((lambda ()                    (lambda () '()) '()))
_____ lambda _____ lambda _____
```

Question 2 : He's a Unix… He's a Unix… He's dead. (10 points)

The Unix file system we use displays directories with slashes between them. For example, our home directory is: `"/home/ff/cs3"`. We'd like to convert that to a sentence of directories by splitting that word `w` into many words in a sentence. Rather than write a splitter specific to `"/"`, we'll write one that can split on any character `c`.

Fortunately, the split char *will always be* the first character, *won't be* the last character, and *will never appear twice in a row* in the input word (e.g., `"//"`).

```
(unixsplit "/" "/home")        ➔ (home)
(unixsplit "/" "/home/ff/cs3") ➔ (home ff cs3)
(unixsplit ":" ":home:ff:cs3") ➔ (home ff cs3) ;; older Macs used ":" not "/"
```

Fill in the blanks to complete `unixsplit` to split a word `w` by the input character `c`. We've started it for you.

```
(define (unixsplit c w)

  (accumulate

    (lambda (ltr ans)

       (cond ((word? ans) (se ltr)) ;; first time through, strip c, return sent
                          ((equal? ltr c) (se "" ans))
             _____________________________________________________________________
                          (se (word ltr (first ans)) (bf ans))
             (else _____________________________________________________________ )))

    (word (bf w) c)))  ;; strip c from beginning, add it to the end
```

Login: `cs3-_____`

Question 3: Magical Mystery Function, step right this way… (10 pts)

```
(define (mystery x)
   (lambda (y)
      (keep x y)))
```

Remember: it's not enough to say the domain or range is simply "a function". You have to describe the domain and range of that function! (…and so on if its domain/range is also a function)

What is the *domain* of `mystery`? (3 pts)

x is a function whose domain is (at least) words and whose range is anything, (but preferably #f at least sometime for mystery to be useful)

What is the *range* of `mystery`? (3 pts)

A function whose domain is words or sentences and whose range is the same as its domain (words or sentences)

Fill in the blanks to complete the interaction with `mystery`. (4 pts).

```
                                  (                    (λ (w) (not (number w))) )

STk> (define numberstripper _______ mystery ____________________________________)
STk> (numberstripper '(msg 4 u 2 i just 8)) ➔ (msg u i just) ;; remove nums
```

Question 4: Aaaaaair ball… Aaaaaair ball… (10 pts)

You play on the Cal basketball team, and have been keeping track of the free throws you've made and missed for a *game* as a word of `1`s (makes) and `0`s (misses). You store your season (every game's results) in a sentence. E.g., if your season were the sentence `(1000 11 "" "00")`, that would mean that you played in 4 games. In the first game (the `1000`), you made one, and missed the next three, in the next game you made two, in the third game you didn't shoot free throws at all, and in the last game you missed twice. We want you to write `worst-game`, to find out the free throw percentage for your *worst game*. You may assume that you shot *at least* 1 free throw this season.

```
(worst-game '(1000 11 "" "00")) ➔ 0     ;; In your last game, you went 0-for-2
(worst-game '(1000 "" 10 ""))   ➔ 0.25  ;; In your first game, you went 1-for-4
(percentage 1000)               ➔ 0.25  ;; A helper; it requires non-empty input
(shot-free-throws? 1000)        ➔ true  ;; This is a helper we wrote for you that
(shot-free-throws? "")          ➔ false ;; returns true when you shot free throws
```

- You **may not** define any additional helper procedures
- You **may not** use `lambda` or `if` or any *explicit* recursion
- You **may only** use higher-order functions (and `percentage`, `shot-free-throws?` and other standard scheme built-in functions, like `+`, `-`, `min`, `max`, etc.)

```
                                        (accumulate + game) ;; ← this also works
                                    (/ (appearances 1 game) (count game))
(define (percentage game)  ____________________________________________________ )

(define (worst-game season)
       (accumulate min (every percentage (keep shot-free-throws? season)))
   ____________________________________________________________________________ )
```

Login: `cs3-_____`

Question 5: Go Bear! (15 points)

You decide to simulate your stock portfolio as it plummets downward.
The function `stock` takes in the starting value `n` and returns a sentence that simulates what happens to it over time. When the stock reaches `1`, you sell it.

```
(define (stock n)
  (se n
      (if (= n 1)
          'sell!
          (stock (if (even? n)
                     (/ n 2)
                     (+ n 1) )))))
```

a) What happens to stocks that were valued at 4? I.e., what will `(stock 4)` return? If it is an error, say what it is. If it is an infinite loop, write "∞ loop". (2 pts)

`(4 2 1 sell!)`

b) What happens to stocks that were valued at 5? I.e., what will `(stock 5)` return? If it is an error, say what it is. If it is an infinite loop, write "∞ loop". (4 pts)

`(5 6 3 4 2 1 sell!)`

c) Now let's do some analysis of a blue chip stock valued at 1,999. What are the first three and last three elements of `(stock 1999)`? Fill in the blanks below. (3 pts)

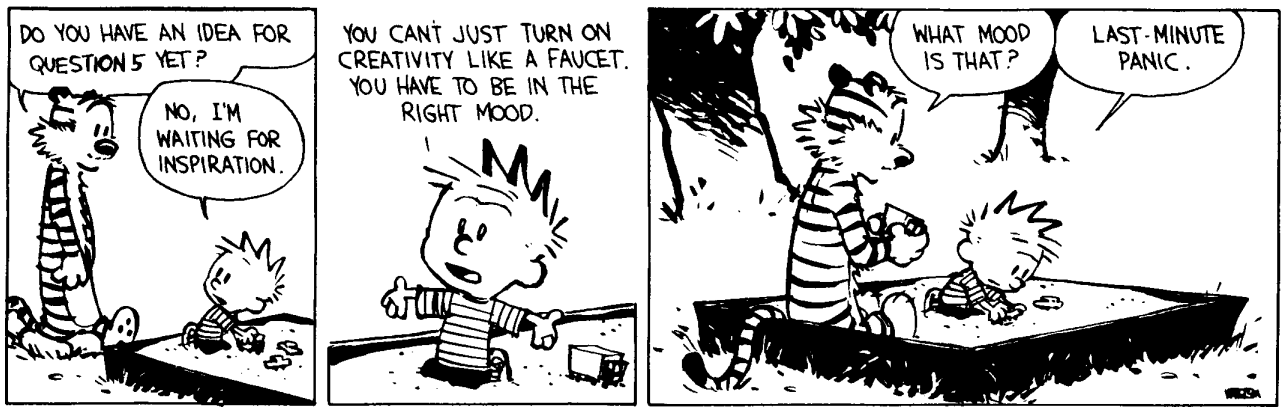
(`1999` `2000` `1000` . . . `2` `1` `sell!`)

d) If we changed the last line from "`(+ n 1)`" to "`(+ n 2)`", what values of `n` would be unaffected (i.e., `stock` would return the same value)? (3 pts)

Powers of 2: 2^0=1, 2^1=2, 2^2=4, etc. because the else line wouldn't be hit

e) For those values of `n` that *would* be affected by the change in (d), what would `stock` return? If it's an error, say what it is. If it is an infinite loop, write "∞ loop". (3 pts)

∞ loop



Login: `cs3-_____`

Question 6 : Does recurs-in `occurs-in?` recursion? No! (20 points)

You want to write the predicate `occurs-in?` from lab to check if a given pattern *pat* occurs in a word *w* as a contiguous subsequence:

```
                pat      w
(occurs-in? 'ello   'hello)   ➔ true
(occurs-in? 'cat    'hello)   ➔ false
(occurs-in? 'hlo    'hello)   ➔ false
(occurs-in? 'hello  ""    )   ➔ false
(occurs-in?  ""     'hello)   ➔ true
```

You try to write `occurs-in?` but it's buggy:

```
(define (occurs-in? pat w)
  (cond
1   ((empty? pat)  true)
2   ((empty? w)   false)
3   ((equal? (first pat) (first w))
4    (occurs-in? (bf pat) (bf w)) )
5   (else
6    (occurs-in? pat (bf w)) )))
```

a) What is the *smallest input* that reveals the bug? (6 pts) That is,

`(occurs-in?` `'ac` `'abc`) returns `true` instead of `false`.

b) Aha! Your very smart friend says … that's because on line `3` you're only testing equality of the *very first* letters. You should change that `cond` clause (lines `3` and `4`) to see if the *entire* `pat` is at the front of `w`! If so, you're done and return `true`, otherwise fall through to the `else` and keep recursing on `w`. Your friend changes line `4`, it's your job to change line `3` (we've split it into `3a` & `3b`) so that `occurs-in?` will work correctly. Line `3a` should check that `pat` is no bigger than `w`. (12 pts)

```
(define (occurs-in? pat w)
  (cond
1   ((empty? pat)  true)
2   ((empty? w)   false)
                (<= (count pattern) (count w))
3a    ((and ____________________________________________________________
                          (( repeated bl (- (count w) (count pattern))) w)
3b          (equal? pat ____________________________________________________ ) )
                          Hint: You might find repeated handy in the line above

4     true) ;; ← your friend changed this line, you change the ones above
5   (else
6    (occurs-in? pat (bf w)) )))
```

c) Does `occurs-in?` employ (circle one) TAIL or EMBEDDED recursion? (2 pt)

TAIL (no embedding of the recursive call)