University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Spring 2001 Instructor: Dan Garcia 2001-04-09

CS 3

Midterm #2

Personal Information

Last name	
First Name	
Student ID Number	
*Lab Section Time & Location **you attend***	
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

- Question 0 (worth 1 point) Please fill in the front and write your name on every page!
- You have two hours to complete this midterm. The midterm 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**.**
- You may always write auxiliary functions for a problem unless they are specifically prohibited 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.
- Please comment on the exam on the right. Rate its difficulty (0 = cake, 5 = impossible), fairness (0 = unfair, 5 = fair), and feel free to add any other comments that come to mind.

Name: _____

Grading Results

Question	*Max. Points*	*Points Given*
0	**1**	
1	**13**	
2	**20**	
3	**10**	
4	**16**	
Total	**60**	

Comments:

- Difficulty (0=easy, 5=hard):
- Fairness (0=unfair, 5=fair):
- Other thoughts?

Name: _____

Question 1 : So one time…at dance camp… (13 points)

You decide to attend a dance camp. The day before you get there you do nothing. When you arrive at the camp, they teach you an interesting dance, which you perform for several days. The dance is as follows:

1) Hop on your left foot the number of days you've been at camp.
2) Do the dance you did yesterday.
3) Hop on both feet the number of days you've been at camp.
4) Do the dance you did yesterday.
5) Hop on your right foot the number of days you've been at camp.

You decide to encode this as a Scheme procedure `dance`, which takes in what day it is and returns a sentence telling you what to do on a particular day. You use a code: "left foot hop" is `L`, "both feet hop" is `B` and "right foot hop" is `R`, and you attach a number to the end of that code to indicate the number of times to do it. So, for example, `R5` means "hop on your right foot 5 times". Even though you may be hopping several times, each code (e.g., `R5`) is considered one "thing-to-do".

```
(define (dance day)           ;; What should I do at dance today?
   (if (= day 0)              ;; before you got to camp
       '()                    ;; do nothing
       (se (word 'L day)      ;; hop on your left foot day times
           (dance (- day 1)) ;; do the dance you did yesterday
           (word 'B day)      ;; hop on both feet day times
           (dance (- day 1)) ;; do the dance you did yesterday
           (word 'R day))))  ;; hop on your right foot day times
```

a) *What do you do* on the first day? I.e., what does `(dance 1)` return? (1 point)

b) *What do you do* on the second day? I.e., what does `(dance 2)` return? (4 points)

c) *How many things* do you do on the third day? I.e., what does `(count (dance 3))` return? (4 points)

d) What is the *90th thing you do* on the 100th day? I.e., what is the 90th word of `(dance 100)`? Hint: look for patterns! (4 points)

Name: _____

Name: _____

Question 2 :Yay bor de bay bor de bor bork bork bork!… (20 points)

The Swedish Chef wants to write a program to simulate the putting-together of foods. For example:

```
: (put-together '(bacon lettuce tomato) 'bread) ➔ baconbreadlettucebreadtomato
: (put-together '(chicken basket) '-in-a-) ➔ chicken-in-a-basket
```

More formally, `put-together` takes in `s`, a sentence of words and `glue`, a glueing word, and returns a word will all the words in `s` put together with `glue`. However, if there's only one thing to put together, it doesn't stick anything in between:

```
: (put-together '(chocolate) 'moose) ➔ chocolate
```

Finally, `put-together` should not have to worry about putting together empty sentences.

a) What pattern is `put-together`? Circle one. (2 points)

MAPPING FINDING COUNTING FILTERING TESTING COMBINING

b) Write `put-together` using *embedded recursion* with **no helper functions and no higher-order functions**. (6 points)

```
(define (put-together s glue)
   (if ____________________________________________________________
       ____________________________________________________________
       ____________________________________________________________ ))
```

c) Write `put-together` using *tail-recursion* with **no higher-order functions**. (6 points)

```
(define (put-together s glue)
```

d) Write `put-together` using *a single higher-order function*, **no helper functions and no recursion** (6 points).

Name: _____

```
(define (put-together s glue)
  ____________________________________________________________ ))
```

Name: _____

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

Name: _____

Question 3 : Driving Miss Calculate… (10 points)

Clarissa Calculate writes a function called `calculator` that takes a sentence of the form `(number operation number operation...number)`, where `plus` and `times` are the only operations, and calculates the result:

```
: (calculator '(5)) ==> 5                          ;; 5
: (calculator '(2 plus 2)) ==> 4                   ;; 2 + 2
: (calculator '(1 plus 3 times 4 plus 1)) ==> 14   ;; 1 + 3 * 4 + 1
: (calculator '(4 times 3 plus 1 plus 1)) ==> 14   ;; 4 * 3 + 1 + 1
```

Notice that `times` is always more important than `plus`. I.e., `(1 plus 2 times 3)` is evaluated as `1 + (2 * 3)`, and **not** `(1 + 2) * 3`. You may assume that `calculator` is always called on valid sentences and is never called on the empty sentence. Unfortunately, Miss Calculate's code has two bugs in it that you need to fix. Consider the following *incorrect* code:

```
(define (second s) (first (bf s)))
(define (third s)  (first (bf (bf s))))

(define (calculator s)
  (cond
1   ((empty? (bf s))
2    0)
3   ((equal? (second s) 'plus)
4    (+ (first s)
5       (calculator (bf (bf s)))))
6   (else
7    (calculator
8     (se (* (first s) (third s))
9         (bf s)
     )))))
```

a) Typing `(calculator '(1 plus 2 plus 3))` gives you `3` when it should return `6`. Replace one line so that `(calculator '(1 plus 2 plus 3))` correctly returns `6`. The code should then work for all sentences which only use `plus`. (5 points)

Replacing line # ____,

with _____

would cause `(calculator '(1 plus 2 plus 3))` to correctly return `6` instead of `3`.

b) There is one remaining bug. Typing `(calculator '(1 plus 2 times 3))` goes into an infinite loop and never returns an answer when it should return `7`. Replace one line so that `(calculator '(1 plus 2 times 3))` correctly returns `7`. After making both bug fixes in parts a and b, `calculator` **should work for all valid input**. (5 points)

Replacing line # ____,

with _____

would cause `(calculator '(1 plus 2 times 3))` to correctly return `7`.

Name: _____

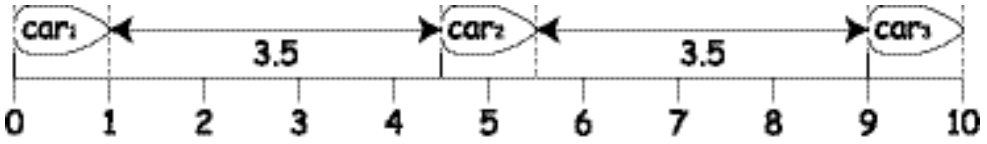
Question 4 : Parking in Berkeley (get it?) (16 points)

Have you ever tried to park on a side street in a Berkeley residential area? People do such a poor job parking that they waste most of the street. How many cars can park on a particular street given that people park randomly?[1] Finding a formula for the average number of cars (all of a given size) that can park on a street (of a given length) isn't easy. However, it's not too hard to simulate (we're going to assume cars don't need any extra "breathing room" space between them to park):

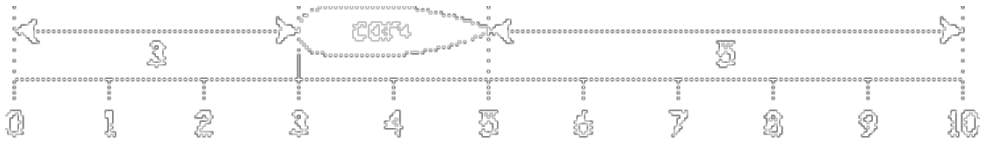
1) See if the section of street you are looking at can fit at least one car. If it can, assume somebody will try to park.
2) Pick a random location for a car to park.
3) Divide the street into two parts: the street behind the car & the street in front.
4) See how many cars can park behind the one that just parked. See how many can park in front of the one that just parked.

Luckily, someone else has done step #1 and #2 above for us. They've provide a semi-predicate called `park?`, which takes the length of a car and the length of the available street. It *randomly* picks a place for the car to park and returns *the location of the car's back end*. If `park?` can't even park one car (i.e., it is called with the length of a car bigger than the length of the street), it returns `#f`. E.g.,

```
(park?  1 10) ==> 0   ;; this car (car1) parked at one end of the street
(park?  1 10) ==> 4.5 ;; this car (car2) parked in the middle of the street
(park?  1 10) ==> 9   ;; this car (car3) parked at the other end of the street
(park? 11 10) ==> #f  ;; this huge car is bigger than the available street
```




```
(park?  2 10) ==> 3   ;; this big car (car4) parked, two spaces (3 and 5) left
```



Write a procedure called `num-parked`, which takes `car`, the length of a car and `street`, the length of the street, and returns the number of cars that were able to park on the street before they ran out of space. *Use no helper functions*. (16 points)

```
(num-parked 1 10) ==> 7 ;; this time, 7 cars all of size 1 were able to park
```

[1] Mathematicians call this the parking problem. It also shows up in fun fields like gene sequencing.

Name: _____

```
(define (num-parked car street)

   (let ( ____________________________________________________ )

      (if ____________________________________________________

          ____________________________________________________

          ____________________________________________________ )))
```