University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2003 Instructor: Kurt Meinz 2003-07-11

CS 61A Midterm #1

Personal Information

First and Last Name	
Your Login	cs61a-__ __
Lab Section Time & Location ***you attend***	
Discussion 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 CS61a who have not taken it yet."	***(please sign)***

Instructions

- Partial credit may be given for incomplete / wrong answers, so please write down as much of the solution as you can.
- Feel free to use any Scheme function that was described in lecture or sections of the textbook we have read without defining it yourself. Do not use functions or constructs that we have not yet covered. Unless specifically prohibited, you are allowed to use helper functions on any problem.
- Please use "true" instead of #t , and "false" instead of #f. We have found that handwritten #t and #f unfortunately look too much alike.
- Please write legibly! If we can't read it, we won't grade it!

Grading Results

Question	*Max. Points*	*Points Earned*
1	**6**	
2	**7**	
3	**6**	
4	**7**	
5	**7**	
6	**7**	
Total	**40**	

Name: ____________________________ **Login:** _____________________

Question 1: Recursion versus Iteration [5 Points]

Consider the function **(remove index lst)** that returns a new list with the index'th element (counting from 1) removed from lst as follows:

>(remove 1 '(a b c)) ➔ (b c)
>(remove 4 '(a b c d)) ➔ (a b c)

You may assume that remove will always be called with index between 1 and the size of the non-empty argument list, inclusive.

Part a:

`Please write remove so it evolves a` **`recursive`** `process:`

```
(define (remove index lst)
```

Part b:

Now, please write remove so it evolves an **iterative** process:

(define (remove index lst)

Name: ________________________ `Login:`________________________

Question 2: Higher-Order Functions `[ 4 Points ]`

Please define a procedure called 'make-keeper' that takes a unary predicate function as an argument and **returns a procedure** that, when invoked on a sentence argument, will keep only those elements that satisfy the predicate. For example:

((make-keeper even?) '(1 2 3 4 5 6)) ➔ (2 4 6)

You may assume that the input to make-keeper will always be a unary function and that the input to the returned procedure will always be a sentence.

Part a:

Please write make-keeper using higher-order functions but no helpers:

(define (make-keeper func)

Part b:

Now, please re-write make-keeper **without** using higher-order functions or helpers (lambda is OK):

(define (make-keeper func)

Question 3: Functions and Higher-Order Procedures `[ 4 Points ]`

Please write a procedure named ‘find-func’ that works as follows:
O find-func takes three arguments: a set of functions, a domain, and a range, each of which is represented as a list.
O find-func will return the one function in the argument list that describes the relation between the given domain and range.
O In other words, find-func will return the one function in func-list that, when applied to each element in the domain, yields the corresponding element in the range.

```
>(find-func (list square double cube) ‘(1 2 3) ‘(2 4 6))
#[Closure for double]
>(find-func (list even? odd? word?) ‘(1 2 3) ‘(#f #t #f))
#[Closure for even?]
```

You may assume that exactly one function will be a perfect match, and you may assume that each function will be valid over each element in the domain.

```
(define (find-func func-list domain range)
```

Question 4: Abstraction `[ 4 Points ]`

Our list implementation has a few weaknesses; namely, common operations like getting the last element of a list and finding the length of a list take linear time. We would like you to provide a new ADT called 'superlist' that will have constant-time length and last procedures.

This ADT will require you to write four operations:
O a 'make-superlist' constructor that takes a non-empty regular list as an argument and returns a superlist.
O a 'suplist-car' procedure that takes a superlist and returns the first element in constant time.
O a 'suplist-length' procedure that takes a superlist and returns the length (# of elements) of the superlist in constant time.
O a 'suplist-last' procedure that takes a superlist and returns the last element of the superlist in constant time.

(make-superlist '(1 2 3 4)) ➔ [Whatever]
(suplist-car (superlist '(1 2 3 4))) ➔ 1
(suplist-length (superlist '(1 2 3 4 5 6 7 8))) ➔ 8
(suplist-last (superlist '(1 2 x))) ➔ x

Hints: 1. When you construct a superlist, you'll want to create a data structure that maintains some extra bookkeeping information (like the number of elements and the last element) alongside the original list. 2. Constructing a superlist can take linear time, and feel free to use the list operations 'listlast' and 'length' on the argument list. ('listlast' is last for lists)

(define (superlist lst)

(define (suplist-car suplst)

(define (suplist-length suplst)

(define (suplist-last suplst)

Question 5: Winning Subsets `[ 4 Points ]`

Please write a procedure named ‘adds-to-n?’ that takes a number n and a list of numbers lst and returns true if and only if there is at least one *winning* subset of lst that adds up to exactly n. For example:

```
(adds-to-n? 7 ‘(3 1 4 2)) ➔  #t   ; 3 + 4 = 7
(adds-to-n? 7 ‘(1 8 1 4 5)) ➔ #t   ; 1 + 1 + 5 = 7
(adds-to-n? 3 ‘(1 8 1 4 5)) ➔ #f
```

Hints:
O What should (adds-to-n?   0   <anylist>) return?
O What should (adds-to-n? <anynonzeronumber> ‘()   ) return?
O Think of the recursive calls as *guesses*: guess that the first element of the list of numbers is in the subset, and then guess that it’s not; if either of these guesses are true, then you know that a winning subset can be formed.

```
(define (adds-to-n? n lst)

  (cond ____________________________________________


        ____________________________________________


        (else (or (adds-to-n ______________________________


                    (adds-to-n ______________________________ )…)
```

Name: ______________________ Login: ______________________

Question 6: Find Dat Bug [5 Points]

Greg wanted to write a procedure that would split a non-empty word into a sentence of consecutive, identical letters as follows:

(split ‘aaabbcdddaa) ➔ (aaa bb c ddd aa)
(split ‘abababab) ➔ (a b a b a b a b)
(split ‘aaa) ➔ (aaa)
(split ‘a) ➔ (a)

Here's what he wrote:

```
1: (define (split wd)
2:  (split-help (first wd) (bf wd)))
3:
4: (define (split-help cur wd)
5:  (cond ((empty? wd) (se))
6:       ((equal? cur (first wd))
7:          (split-help (word cur (first wd)) (bf wd))))
8:       (else
9:          (se cur (split-help (first wd) (bf wd))))))
```

There are two bugs in Greg's code. We would like you to find and fix them. **Hint:** None of the bugs are syntax errors (mismatched parens, misspelled arguments, etc.)

Part a:

What does (split ‘abc) return? ____________

On which line number is the bug that causes this error? _______

What should the fixed line say?

____

Part b:

There is another bug. On which line is this bug? _____________

What should the fixed line say?