University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2002 Instructor: Kurt Meinz 2002-07-12

CS 61A Midterm #1

Personal Information

First and Last Name	
Your Login	cs61a-__ __
Logins of Group Members	cs61a-__ __ cs61a-__ __ 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	**5**	
2	**5**	
3	**5**	
4	**5**	
5	**5**	
6	**5**	
7	**5**	
8	**5**	
Total	**40**	

Name: _____ Login: _____

Question 1: Applicative and Normal Order [4 Points]

Part a:

Consider this function:

```
(define (greg x y z)
  (z (* x y)))
```

How many times is the '*' procedure invoked when

```
(greg 3 (* 2 2) (lambda (x) (* x x x)))
```

is evaluated in normal order? _____

How many in applicative order? _____

Part b:

Consider this expression:

```
(let ((a (* 2 3)) (b (* 6 7)))
   (if (= a b)
      (* a b)
      b))
```

How many times is '*' invoked if the order of evaluation is applicative?

_____

How many if the order is normal? _____

Name: _____ **Login:**_____

Question 2: Functions **[4 Points]**

Consider the following function:

```
(define (jeffified? a b c)
  (cond ((equal? a b) #t)
        ((pair? c) #t)
        ((odd? (+ a b c)) (if (equal? (- a b) -2)
                              #t
                              #f))
        (else #f)))
```

Please re-write the function so that it has the same behavior but does not use 'cond' or 'if'. You may not use helper functions, but you are encouraged to use the logical functions 'and', 'or', and 'not'.

Name: _____ **Login:**_____

Question 3: Recursion **[4 Points]**

Write a procedure named 'running-total' that takes a non-empty **sentence** of numbers as an argument and returns another sentence that contains the running total.

In other words,
O The first number in the resulting sentence should be the first number of the argument sentence,
O The second number in the resulting sentence should be the sum of the first and second numbers of the original sentence,
O The third number in the resulting sentence should be the sum of the first, second, and third numbers of the original sentence, and so on.

For example:

```
> (running-total '(1 2 3 4))
(1 3 6 10)
> (running-total '(-5 0 -22 18 55))
(-5 -5 -27 -9 46)
```

Name: _____ **Login:**_____

Question 4: Let **[4 Points]**

Consider the following snippet of scheme code:

```
(let ((x 3) (y 4))
  (let ((x 5) (y (+ x 3)))
    (let ((x (+ x 7)))
      (+ x y))))
```

Part a:

What is the return value of the expression? If you think it is an error, write 'error' and give a concise description of the cause of the error.

_____

Part b:

Re-write the snippet by replacing all of the let expressions with their lambda equivalents.

Name: _____ **Login:**_____

Question 5: Lists **[4 Points]**

Erwin wanted to implement the 'list?' predicate. Here is his first attempt:

```
(define (erwins-list? l)
  (or (null? l)
      (erwins-list? (cdr l))))
```

Jane thinks that this procedure will fail if someone tries to evaluate the following:

```
(erwins-list? (cons 3 (cons 4 5)))
```

Part a:

Is Jane correct? _____

If yes, why? If no, why not? (Be concise!)

Part b:

Fix Erwin's procedure in the space provided below. We are looking for the smallest and simplest possible change such that the procedure will work for any input whatsoever.

Name: _____ **Login:**_____

Question 6: Higher Order Procs **[5 Points]**

Question 2.6 of SICP (page 93) makes reference to the idea of implementing numbers using lambdas – much like we implemented paris via lectures in lecture. Consider the following implementation of 'kurt-numerals':

```
(define zero (lambda (x) x))

(define (add-one kurt-num)
   (lambda (x) kurt-num))
```

Part a:

As specifically as possible describe in English the object that is the return value of this expression:

```
(add-one zero)
```

Part b:

Please write a procedure named 'show-kurt-val' that takes a (well-defined) kurt-numeral as an argument and returns the number that that kurt-numeral corresponds to.

For example:

```
> (show-kurt-val (add-one (add-one (add-one zero)))
3
```

Name: _____ **Login:**_____

Question 7: Abstraction **[5 Points]**

Consider these six mystery functions:

```
(define (func-1 x)
  (x 'false))
```
func-1:_____

```
(define (func-2 x)
  (not x))
```
func-2:_____

```
(define (func-3 x)
  (x #f)
```
func-3:_____

```
(define (func-4 p q)
  (lambda (y)
     (if y q p)))
```
func-4:_____

```
(define (func-5 x y)
  (if y
      (func-4
        (x (func-3 y))
        (func-5 x (func-1 y)))
      #f))
```
func-5:_____

```
(define (func-6 x y)
   (cond ((func-2 y) #f)
         ((x (func-3 y))
          (func-4 (func-3 y) (func-6 x (func-1 y))))
         (else
           (func-6 x (func-1 y)))))
```
func-6:_____

Part a:

When taken together, these six functions constitute a partial implementation of a datatype that we have used in this class.

What is this datatype? _____

Part b:

Please label each mystery function with the name that we usually use for it. For example, if you think that func-5 is an implementation of 'sentence', then write 'sentence' on the line next to func-5.

Name: _____ **Login:**_____

Question 8: Orders of Growth **[5 Points]**

Give Big-Theta bounds for the time complexity of the following procedures as a function of the input size. You may assume that 'foo' is a globally-defined procedure that runs in Big-Theta(n) time.

Part a:

```
(define (ilya n)                          Big-Theta(        )
  (if (< n 100000)
      34
      (ilya (- n 1))))
```

Part b:

```
(define (ilya n)                          Big-Theta(        )
  (if (< n 100000)
      34
      (+ (foo (- n 1))
         (foo (- n 2))
         (foo (- n 3)))))
```

Part c:

```
(define (ilya n)                          Big-Theta(        )
  (if (< n 100000)
      34
      (+ (ilya (- n 1))
         (ilya (- n 2))
         (ilya (- n 3)))))
```

Part d:

```
(define (ilya n)                          Big-Theta(        )
  (if (< n 100000)
      34
      (+ (foo  (- n 1))
         (foo  (- n 2))
         (ilya (- n 3)))))
```