CPSC 128 Midterm
October 28th, 2003

Family Name: _____

Given Name: _____

Student ID#: _____

- You have 80 minutes to write this test.
- The test consists of 5 questions.
- There are a total of 46 marks.
- No notes or electronic equipment (calculators, laptops, cell phones) are allowed.
- Use the back of the pages for scratch work.
- Good luck!

Question	**Max Marks**	**Marks Earned**
1	8	
2	14	
3	2	
4	12	
5	10	
Total	**46**	

Question 1 (8 marks)
a) (2 marks)
How does Scheme keep track of defined names?

It stores the names and associated bound values in a table in memory called the environment.

b) (2 marks)
What is a helper procedure, and why do we use helpers?

A procedure that solves a small portion of hte problem. We use them to divide the solution into parts so that it's easier to implement the solution, and to read and maintain and test the code.

c) (2 marks)
Why do we have both let* and letrec? Why can't we just use letrec all the time?

You can't substitute letrec for let* because letrec does not guarantee left to right binding the way let* does.

d) (2 marks)
The following will produce an error. Why? (be specific)

```
(define a 4)
(quotient (a 3))
```

It is attempting to apply procedure a to parameter 3. a evaluates to 4, not a procedure, so you will get the error "4 is not a procedure"

Question 2 (14 marks)
For each of the following groups of forms, write down the value of the last form of the group. In no case will an error occur. When the value of the last form is a pair (or list), draw the box-and-arrow diagram for it. You may choose to explain how you obtained your answers for consideration in grading.

a) (2 marks)

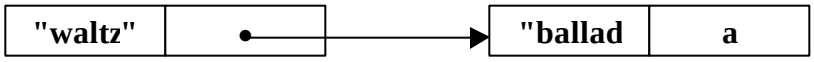
```
(define proc
  (lambda (a b)
    (let* ((a 5) (c (* a a)))
      (+ a b c))))
(proc 2 1)
```

31

b) (4 marks)

```
(define a "waltz")
(define b "ballad")
(define c (cons (list a b) 'a))
(define d (append (list-ref c 0) (cdr c)))
d
```

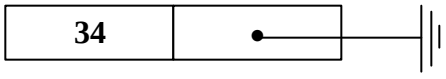
("waltz" "ballad" . a)



c) (4 marks)

```
(define proc
  (lambda (m the-list)
    (letrec ((helper (lambda (n the-list)
                       (if (null? the-list)
                         '()
                         (if (and (even? (car the-list))
                                  (> (remainder (car the-list) n) 0))
                           (cons (car the-list)
                                 (helper (+ 1 n) (cdr the-list)))
                           (helper n (cdr the-list)))))))
      (helper m the-list))))
(proc 3 '(5 17 34 16))
```

(34)



d) (4 marks)

```
(define a (lambda (proc x) (proc (car x) (cdr x))))
(define b (cons "Bugs" "Bunny"))
(a (lambda (s1 s2) (string-append s1 (substring s2 4 5))) b)
```

"Bugsy"

Question 3 (2 marks)

If the time complexity of (bar m) is O(logm), what is the time complexity of foo?

```
(define foo
  (lambda (n m)
    (if (<= n 0)
        0
        (+ (bar m) (bar (- m 1))
           (foo (quotient n 5) m)))))
```

$O(\log_n \log_m)$

Question 4 (12 marks)

Examine the following interface for the Abstract Data Type called Film. This data type stores information about a Film, including its title, genre, and rating:

`(make-film title description rating)`	constructs a new film data structure
`(film-title film)`	returns the title of the film
`(film-genre film)`	returns the genre of the film
`(film-rating film)`	returns the rating of the film

a) (2 marks)

Using this abstract interface, write a Scheme form to define a list of the following 3 films.

Title	Genre	Rating
"Bend it Like Beckham"	"Comedy"	"PG"
"Good Boy!"	"Family"	"G"
"Lost in Translation"	"Drama"	"14A"

```
(list (make-film "Bend it Like Beckham" "Comedy" "PG")
      (make-film "Good Boy!" "Family" "G")
      (make-film "Lost in Translation" "Drama" "14A"))
```

b) (10 marks)
Using only this abstract interface, write a Scheme procedure that takes an arbitrary list of Films and returns a new list containing only the titles of films with the rating "G". So for example, if you passed this procedure the list you created in part (a), it would return the list ("Good Boy!"). You may assume that any procedure discussed in class has already been implemented and may be used freely. (Your solution does not have to take advantage of this assumption, although it would simplify it) Hint: do not assume you know how the Film data structure is implemented!

```
(define g-titles
  (lambda (films)
    (map film-title
      (filter (lambda film)
                (equal? (film-rating film) "G"))
              films))))
```

Question 5 (10 marks)
Write a procedure `deep-sum`, which sums up the values of all elements in all levels of a list. For example, the application `(deep-sum '(1 (5 17 (2 14) 18) 32 1))` returns the value `90`.

```
(define deep-sum
  (lambda (the-list)
    (if (null? the-list)
      0
      (if (pair? (car the-list)
        (+ (deep-sum (car the-list)) (deep-sum (cdr the-list)))
        (+ (car the-list) (deep-sum (cdr the-list)))))))
```

Scheme Primitives

Here are some scheme primitives you may find useful in this test:

Equality

(equal? *obj1 obj2*)	returns #t if *obj1* and *obj2* have the same structure and value, #f otherwise

Pairs and Lists

(pair? *obj*)	returns #t if *obj* is a pair, #f otherwise
(cons *obj1 obj2*)	returns a new pair whose car contains *obj1* and whose cdr contains *obj2*
(car *pair*)	returns the first element in *pair*
(cdr *pair*)	returns the second element in *pair*
(null? *obj*)	returns #t if *obj* is null, #f otherwise
(list? *obj*)	returns #t if *obj* is a proper list, #f otherwise
(list *obj1* …)	returns a list made up of each *obj*
(length *list*)	returns the number of pairs in *list*
(append *list1* …)	returns a list consisting of the elements of the first *list* followed by those of the second *list*, etc.
(list-tail *list n*)	returns the pair of *list* obtained by applying *n* cdrs to *list*
(list-ref *list n*)	returns the *n*th element of *list*

Numbers

- (zero? *num*):	returns #t if *num* is 0, #f otherwise
- (odd? *num*):	returns #t if *num* is odd, #f otherwise
- (even? *num*):	returns #t if *num* is even, #f otherwise
- (quotient *numer denom*):	returns the truncated integer *numer* / *denom*
- (remainder *numer denom*):	returns the remainder of the integer *numer* / *denom*
- (sqrt *num*):	returns the square root of *num*

Strings

- (string? *var*):	returns #t if *var* is a string, #f otherwise
- (string=? *str1 str2*):	returns #t if *str1* and *str2* have the same value, #f otherwise
- (string *ch*):	returns a string consisting of the single character *ch*
- (string-length *str*):	returns the length of the string *str*
- (string-ref *str pos*):	returns the character at position *pos* of string *str*
- (substring *str start end*):	returns a new string containing the characters of *str* from position *start* to *end*-1
- (string-append *str1 str2*):	returns a new string made by appending the characters of string *str2* at the end of *str1*

Procedures and Mapping

(map *proc list*)	applies *proc* to each element of *list* and returns a list of the results