CS61A, Fall/1999
Midterm #1
Professor Brian Harvey

Problem #1 (3 points)

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just say "error"; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just say "procedure"; you don't have to show the form in which Scheme prints procedures. **Also, draw a box and pointer diagram of the value produced by each expression.**

```
(let ((x '(a b c)))
  (cons (cdr x) (car x)))
```

```
(cdar '((1 2 3) (4 5 6)))
```

```
(append (cons '(a) '(b)) (list 'c 'd))
```

Problem #2 (2 points)

Suppose that you have a procedure (f n) that requires time theta(*n*) and a procedure (g n) that requires theta(*n*^2). What is the order of growth required for a program to compute (* (f n) (g n))? **There may be more than one correct answer; check all that are correct.**

_____ A. theta(n)

_____ B. theta(n^2)

_____ C. theta(n + n^2)

_____ D. theta(n^3)

Problem #3 (2 points)

Given the primitive procedure *random* discussed in lecture, and the procedure *square* defined as follows:

```
(define (square x)
    (* x x))
```

Which of the following expressions may have a different value depending on whether applicative order or normal order is used? **There may be more than one correct answer; check all that are correct.**

_____ A. (random 10)

_____ B. (* (random 10) (random 10))

_____ C. (square (random 10))

_____ D. (random (square 10))

Problem #4 (4 points)

This question concerns the twenty-one game used in the first programming project. Now that we know about data abstraction, we want to improve the project by using an abstract data type for cards, with the following constructor and selector:

```
(define make-card word)
(define card-rank butlast)
(define card-suit last)
```

Respect the data abstraction in the procedures you write -- use the constructor and selectors wherever appropriate.

(a) Write the strategy procedure *want-jack* that asks for an additional card unless the player's hand contains a jack. (Yes, this is a very bad strategy.)

(b) Write a procedure *want-rank* that takes a rank as its argument and returns a strategy procedure that asks

for an additional card unless the player's hand contains a card of that rank. So the expression *(want-rank 'j)* should return a strategy equivalent to *want-jack* above.

Problem #5 (4 points)

Consider the abstract data type Crisp that includes three components: a Snap, a Crackle, and a Pop. The constructor for this ADT is:

```
(define (make-crisp s c p)
   (cons (list sc) p))
```

(a) Write the three selectors *snap, crackle, pop* that take a Crisp as argument and return the appropriate component.

(b) Write a procedure *rotate-crisp* that takes a Crisp as its argument (hereafter called c) and returns a new Crisp whose Snap is the Crackle of c, whose Crackle is the Pop of c, and whose Pop is the Snap of c. **Respect the data abstraction.**

Problem #6 (4 points)

(a) Complete the following procedure *sentence?* that takes any Scheme value as its argument and returns #T if the argument is a sentence, or #F otherwise. In order to be a sentence, the argument must be a list, and every element of the list must be a word. You may use the primitive procedure *word?* in your solution.

```
(define (sentence? x)
   (cond ((null? x) __________)
         ((not (pair? x)) __________)
         (_____________________________________ #F)
         (else _____________________________________)))
```

(b) Write a procedure *list-of-sentences?* that takes any Scheme value as its argument, and returns #T if the argument is a list of sentences, or #F otherwise. In order to be a list of sentences, the argument must be a list,

and every element of the list must be a sentence.

(c) Write a procedure *list-of-whatevers?* that takes two arguments;the first must be a predicate function of one argument, and the second can be any Scheme value. It should return #T if the second argument is a list in which every element satisfies the predicate, or #F otherwise.