

INSTRUCTIONS

- You have 1 hour to complete the exam.
- The exam is closed book, closed notes, closed computer, and closed calculator.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation.

| | |
|---|--|
| Last name | |
| First name | |
| SID | |
| BearFacts email (_@berkeley.edu) | |
| <i>All the work on this exam is my own. (please sign)</i> | |

1. (12 points) Scheme

What will Scheme output? Write your answer as a Scheme value. If evaluation causes an error, write ERROR. If evaluation would take forever, write FOREVER. Each example is independent of the previous examples.

(a) (2 pt) `(car (cdr (list 1 2 3)))`

2

(b) (2 pt) `(cdr (cdr (cons 1 (cons (list 2) (list 3)))))`

(3)

(c) (2 pt) `((lambda (x y) (y x)) '(car (1 2)) cdr)`

((1 2))

(d) (2 pt)

```
(define x (mu (y) (x y)))
(define (f y x) (y 3))
(f x (lambda (x) (* x x)))
```

9

(e) (2 pt)

```
(define (g a b) (if (zero? a) 0 (+ a (b (- a 2) b))))
(((lambda (f) (lambda (x) (f x f))) g) 10)
```

30

(f) (2 pt)

```
(define (g a b) (if (zero? a) 0 (+ a (b (- a 2) b))))
(((lambda (f) (lambda (x) (f x f))) g) 11)
```

Forever (error OK because of recursion depth limit)

2. (6 points) Interpreter

For the following expressions, how many times will `scheme_eval` be called? Assume that you are using the non-tail-recursive implementation of Scheme that you developed in your project.

| Expression | # <code>scheme_eval</code> |
|--|----------------------------|
| <code>(define (square x) (* x x))</code> | 1 |
| <code>(+ (square 4) 1)</code> | 10 |
| <code>(car '(foo bar baz))</code> | 3 |

3. (14 points) Define

- (a) (6 pt) Define a function `assoc` that takes a list of two-element lists and a symbol as input. It outputs a list of all the two-element lists that have the symbol as their first element. Use the builtin procedure `equal?` to compare symbols.

For example, here are some sample calls.

```
> (assoc '((foo . bar) (foo . baz) (garply . xyzzy)) 'foo)
((foo . bar) (foo . baz))
> (assoc '((foo . bar) (foo . baz) (garply . xyzzy)) 'garply)
((garply . xyzzy))
> (assoc '((foo . bar) (foo . baz) (garply . xyzzy)) 'xyzzy)
()
```

```
(define (assoc s sym)
  (if (null? s)
      s
      (if (equal? sym (car (car s)))
          (cons (car s) (assoc (cdr s) sym))
          (assoc (cdr s) sym))))
```

```
(assoc '((foo . bar) (foo . baz) (garply . xyzzy)) 'foo)
(assoc '((foo . bar) (foo . baz) (garply . xyzzy)) 'garply)
(assoc '((foo . bar) (foo . baz) (garply . xyzzy)) 'xyzzy)
```

- (b) (8 pt) Define a function `sums` that takes a non-empty list of positive integers in increasing order with no repeats and outputs a list of all the possible results of summing a non-empty subset of them. The resulting list should also be in increasing order with no repeats.

For example, here are some sample calls.

```
> (sums '(1 2 4 8))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
> (sums '(1 4 8))
(1 4 5 8 9 12 13)
> (sums '(1 3 4))
(1 3 4 5 7 8)

(define (sums s)
  (define (f s)
    (if (null? s) '(0)
        (let ((cdr-sums (f (cdr s))))
          (merge cdr-sums
                  (map (lambda (x) (+ x (car s))) cdr-sums))))))
  (cdr (f s)))

(define (merge a b)
  (cond ((null? a) b)
        ((null? b) a)
        ((> (car a) (car b)) (merge b a))
        ((= (car a) (car b)) (merge (cdr a) b)) ; OR (cons (car a) (merge (cdr a) (cdr b)))
        (else (cons (car a) (merge (cdr a) b)))))

(merge '(1 3 5 7) '(3 4 5 6))
; expect (1 3 4 5 6 7)
(sum '(1 2 4 8))
; expect (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(sum '(1 4 8))
; expect (1 4 5 8 9 12 13)
(sum '(1 3 4))
; expect (1 3 4 5 7 8)
```

4. (8 points) Flight Home

Select all of the one-stop flights from SF and their total time. The time of a one-stop flight is the sum of the times of each leg and the transfer time of the connecting airport.

```
create table airports as
select "SF" as city, 20 as transfer union
select "LA"      , 30      union
select "DC"      , 40      union
select "NY"      , 50      union
select "Vegas"   , 60      union
select "Reno"    , 10;

create table flights as
select "SF" as start, "LA" as end, 70 as time union
select "SF"      , "Reno"      , 50      union
select "SF"      , "Vegas"     , 90      union
select "LA"      , "Vegas"     , 30      union
select "LA"      , "DC"        , 200     union
select "DC"      , "NY"        , 100     union
select "Reno"    , "NY"        , 250     union
select "Vegas"   , "NY"        , 280;

select b.start as connection,
       b.end as destination,
       a.time + b.time + transfer as time
  from flights as a, flights as b, airports
 where a.end = b.start and a.start = "SF" and b.start = city;
```

The expected output is:

```
LA|DC|300
LA|Vegas|130
Reno|NY|310
Vegas|NY|430
```

To receive full credit, you must write your statement so that it would work correctly even if the contents of the airports and flights tables were to change.