NAME

CS 61A
Fall 2002

Midterm #3

L. Rowe

This examination is open book and notes, but closed friends! Answer all questions in the space provided. Some questions have procedures with blank lines where you are to give an answer. You are to provide one expression for each blank line. If you do not understand a question, please ask the proctor for clarification.

Circle Login

-a	a
-b	b
-c	c
-d	d
-e	e
-f	f
-g	g
-h	h
-i	i
-j	j
-k	k
-l	l
-m	m
-n	n
-o	o
-p	p
-q	q
-r	r
-s	s
-t	t
-u	u
-v	v
-w	w
-x	x
-y	y
-z	z
-1	

Question	Score	Total Possible
1.		(10 possible)
2.		(10 possible)
3.		(30 possible)
4.		(20 possible)
5.		(20 possible)
6.		(10 possible)
TOTA		100 possible)

Oath

I certify that I am the student whose name appears above.

Signature: ________

Student ID ________

Seating

On my left: in: CS 61a - ________

On my right: in: ________

1. (10 points, 1 point each part) Answer the following true/false questions:

T F A procedure written using functional programming always returns the same value when called with the same actual arguments.

T F The class of the **Object** class is **Object** in the class hierarchy in a conventional OOP language as discussed in lecture.

T F Every class is represented by an instance at run-time.

T F A class variable is shared by all instances of the class.

T F The execution of procedures **p1** and **p2** in

```
(parallel-execute (s p1) (s p2))
```

using a serializer **s** produces the same result as executing

```
(begin (p1) (p2))
```

T F The Scheme stream abstraction uses a procedural representation that allows a possibly infinite sequence of values to be represented.

T F An environment, *E1*, is created when calling procedure **p**. The static and dynamic links for *E1* point to the same environment unless **p** was returned as the value of another procedure or p was passed to the procedure as an explicit argument.

T F Procedures that take a variable number of arguments cannot be written in Scheme.

T F An environment is created when a procedure is called and deleted when the procedure returns.

T F A procedure might change state (i.e., use imperative programming) even though it does not make a direct call on **set!**, **set-car!**, or **set-cdr!**.

2. (10 points) Consider the following definitions:

```
(define balance 100)
(define (withdraw amount)
        (set! balance (- balance amount)))
```

We know that if we execute the following code, the final value of **balance** can be 70, 80 or 90. Also, there is no possibility of a deadlock when executing these procedures in parallel:

```
(parallel-execute (lambda () (withdraw 10))
                  (lambda () (withdraw 20)))
```

Now consider these definitions:

```
(define flag1 #f)
(define flag2 #f)
(define balance 100)
(define (withdraw amount)
        (set! balance (- balance amount)))
(define (test-flag1)
        (if flag1 (test-flag1) 'okay))
(define (test-flag2)
        (if flag2 (test-flag2) 'okay))
```

Suppose we then execute:

```
(parallel-execute (lambda ()
                     (set! flag1 #t)
                     (test-flag2)
                     (withdraw 10)
                     (set! flag1 #f))
                  (lambda ()
                     (set! flag2 #t)
                     (test-flag1)
                     (withdraw 20)
                     (set! flag2 #f)))
```

(a) (1 point) Is it possible for both the processes to be executing their calls to withdraw at the same time? Circle your answer:

YES NO

(b) (3 points) Circle all values of balance that are possible after this code fragment executes:

70 80 90

(c) (1 point) Is there a possibility of a deadlock here?

YES NO

Consider the following code along with the definitions of **balance**, **withdraw**, **flag1**, **flag2**, **test-flag1**, and **test-flag2** given above:

```
(parallel-execute (lambda ()
                     (test-flag2)
                     (set! flag1 #t)
                     (withdraw 10)
                     (set! flag1 #f))
                  (lambda ()
                     (test-flag1)
                     (set! flag2 #t)
                     (withdraw 20)
                     (set! flag2 #f)))
```

(d) (1 point) Is it possible for both the processes to be executing their calls to withdraw at the same time?

YES NO

(e) (3 points) Circle all values of balance that are possible after this code fragment executes:

70 80 90

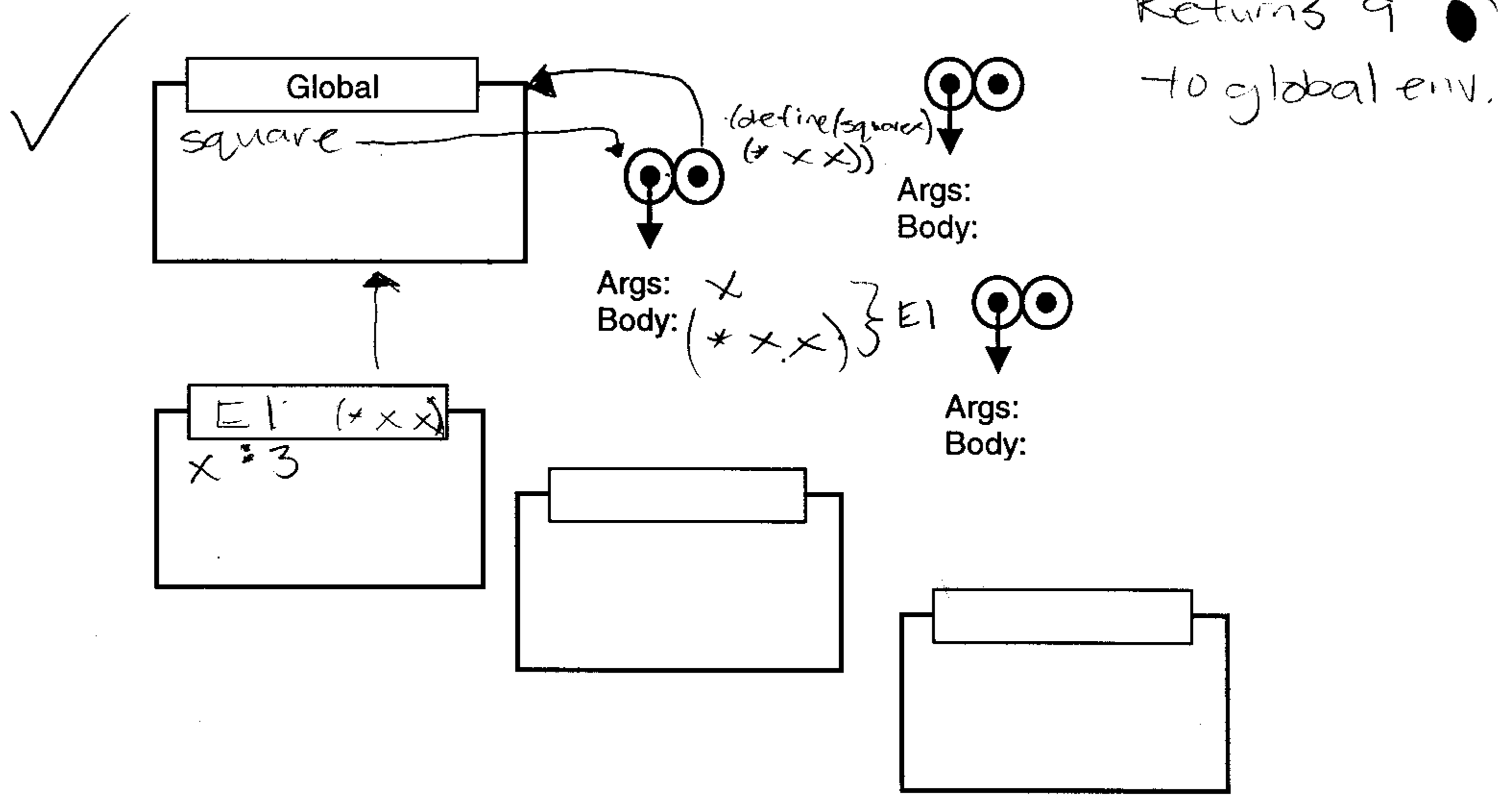
(f) (1 point) Is there a possibility of a deadlock here?

YES NO

3. (30 points) You are to draw environment diagrams for this question. We will give you a series of Scheme expressions and ask you to draw the diagrams after the expressions have been executed. You will be given a template to draw the diagrams that includes boxes for environments and double-bubbles for procedure definitions. You are to show: 1) what procedure execution created the environment, 2) what variables are bound in the environment and the value bound to the variable, and 3) the static link for the environment. For procedure values, you must show the formal argument list, the body of the procedure, and the defining environment. Not all boxes and double-bubbles need be used in the answer, and you can add boxes and double-bubbles if there are not enough.

(a) (8 points) Show the environment after the following code is executed:

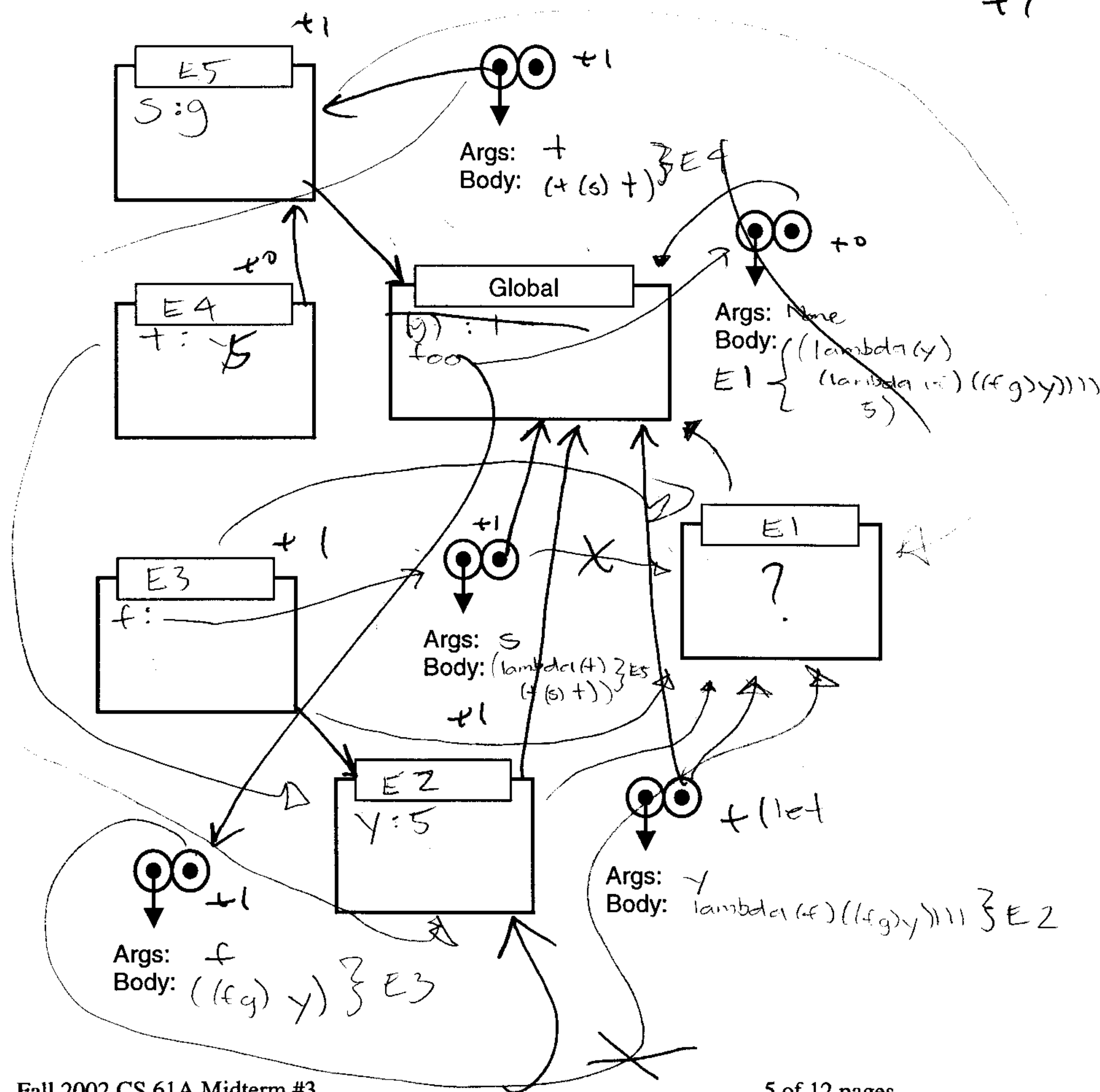
```
(define (square x) (* x x))
(square 3)
```



(b) (2 point) What value is printed by the following code?

```
(define (g) 1)

(define foo
  (let ((y 5))
    (lambda (f) ((f g) y))))

(foo (lambda (s) (lambda (t) (+ (s) t))))
```

(c) (20 points) Show the environment after the code in part b) is executed:



4. (20 points) Numbers can be represented in different bases. We are taught base 10 when we learn basic arithmetic. Computers, on the other hand, use base 2 to represent numbers and implement arithmetic.

   (a) (12 points) Following are two packages that implement addition and subtraction in base 2 and base 10. Fill-in-the-blanks in the code so that all numbers returned by functions in these packages are tagged with **base2** or **base10** as appropriate, and the procedures for operating on these data types (i.e., **add**, **sub**, and **make-num**) are entered into the global table using **put** and retrieved using **get**, which are defined below.

```
(define (install-base-2-package)
  ;; private procedures
  (define (base2to10 num_2)
     ; procedure to convert base 2 numbers to base 10
     )

  (define (base10to2 num_10)
     ; procedure to convert base 10 numbers to base 2
     )

  ;; public functions
  (define (make-base-2-num num_10)
    (base10to2 num_10))
  (define (add num1_2 num2_2)
    (base10to2 (+ (base2to10 num1_2) (base2to10 num2_2))))
  (define (sub num1_2 num2_2)
    (base10to2 (- (base2to10 num1_2) (base2to10 num2_2))))
  ;; install procedures
  (define (tag x) (attach-tag 'base2 x))
  (put 'make-base-2-num '(base 2) make-base-2-num)
  (put 'add '(base2) add)
  (put 'sub '(base2) sub)

   'done)
(define (install-base-10-package)
  ;; install procedures
  (define (tag x) (attach-tag 'base10 x))
  (put 'make-base-10-num '(base 10) make-base-10-num)
  (put 'add '(base10) add +)
  (put 'sub '(base10) sub -)

   'done)
```

You can use the following two procedures to modify the table:

```
(define (put op type item)
  ; puts the procedure item into the table for
  ; operation op and type type
)

(define (get op type)
  ; gets the procedure for operation op on type type
)
```

(b) (2 points) Fill-in the following table to show what mappings have been defined after the above two lines of code are executed. (install-base-2-package)
" " 10 "

Types

-1

not function names

Operations	base2	base10
add	add-base2	add-base10
sub	sub-base2	sub-base10
make-num	make-num-base2	make-num-base10

(c) (6 points) Fill-in the code to make the numbers five and ten in base 2 and add them to get fifteen in base 2. Remember these procedures are being evaluated in the global environment.

ok

```
(define five_2
((get 'make-num '(base2)) 5)                        )

(define ten_2
((get 'make-num '(base2)) 10)                       )

(define fifteen_2
((get 'add '(base2)) five-2 ten-2))
```

(14)

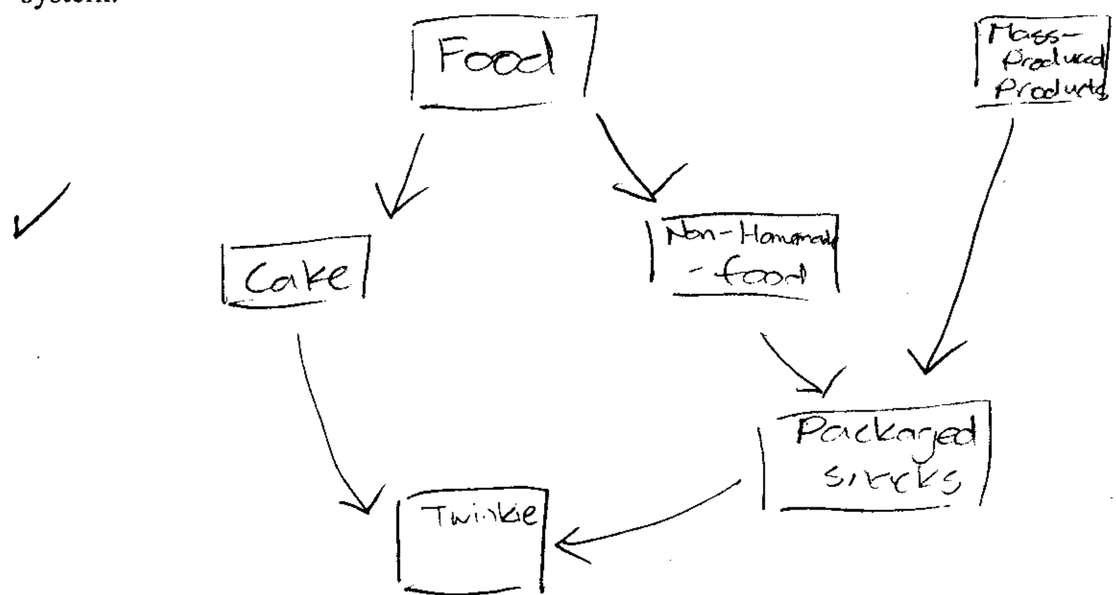
5. (20 points) Consider the following class and procedure definitions:

```
(define-class (Food name)
   (instance-vars (Cook-history '() ))
   (class-vars (all-food '())
   (initialize (set! all-food (cons self all-food)))
   (method (cook style)
           (set! Cook-history (cons style Cook-history)))
   (method (food-name)
           name))

(define-class (Cake name)
   (parent (Food name))
   (instance-vars (color 'brown))
   (initialize (ask self 'cook 'bake)))

(define-class (Non-homemade-food name)
   (parent (Food (word 'Yum name)))
   (instance-vars (popularity 20))
   (method (cook style)
           (append! (ask self 'Cook-history) (list style))
           (ask self 'Cook-history))
   (method (food-name)
           (usual 'food-name)))

(define-class (Mass-produced-products name company)
   (instance-vars (popularity 10))
   (class-vars (all-products '()))
   (initialize (set! all-products (cons self all-products))))

(define-class (Packaged-snacks name company)
   (parent (Mass-produced-products name company)
           (Non-homemade-food name))
   (instance-vars (plastic-wrapped #t))
   (method (unwrap)
           (set! plastic-wrapped #f))
   (method (food-name)
           (usual 'food-name)))

(define-class (Twinkie name)
   (parent (Packaged-snacks name 'Hostess)
           (Cake (word 'Tasty name)))
   (class-vars (color 'yellow))
   (method (what-am-i)
           (usual 'name))
   (method (name-change newname)
           (set! name newname)
           name)
   (method (food-name)
           (usual 'food-name))
   (method (make-moldy)
           (set! color 'green)))
```

(cook freeze)

```
; procedure used in method in Non-homemade-food
(define (append! x y)
  (define (last-pair a)
    (if (null? (cdr a))
        a
        (last-pair (cdr a))))
  (set-cdr! (last-pair x) y)
  x)
```

(a) (6 points) Draw the class hierarchy for the classes defined above. You do not need to show the Object and Class classes, just the classes defined using the Scheme OOP system.



(b) (7 points, 1 point each) Below is a sequence of statements typed into Scheme after loading the classes defined above. Fill-in-the-blanks with the return value for the immediately preceding statement. We have intentionally omitted the return values for some statements, so as not to give away the answer.

```
> (define mytwink (instantiate Twinkie 'mytwink))
> (ask mytwink 'name-change 'yummy)
> (ask mytwink 'what-am-i)
```

mytwink

```
> (define yourtwink (instantiate Twinkie 'yourtwink))
> (ask yourtwink 'make-moldy)
> (ask mytwink 'color)
```

green

```
> (ask mytwink 'Cook-history)
```

() (bake)

```
> (ask mytwink 'cook 'freeze)
> (ask mytwink 'cook 'batter)
```

((freeze) (batter))

```
> (define a-banana (instantiate Food 'banana)
> (ask a-banana 'cook 'fried)
> (ask a-banana 'Cook-history)
```

(fried)

```
> (define cupcake (instatiate Cake 'cupcake)
> (ask cupcake 'food-name)
```

cupcake

```
> (ask mytwink 'food-name)
```

mytwink

(c) (7 points) We want to define a method, named **n_usual**, that is similar to the predefined usual method. The **n_usual** method looks for methods and variables in the nth level superclass as opposed to the immediate superclass. The class of an instance is the 0th level ancestor, the immediate parent of this class is the 1st level ancestor, and so forth.

The **n_usual** method takes two arguments: 1) **level** – the level to be examined and 2) **message** – the name of a method that takes no arguments. For example,

```
(define-class (A name)
   (method (n_usual n message) ...))

(define-class (B name)
   (parent (A (word 'i name))) ...
   (method (n_usual n message) ...))

(define-class (C name)
  (parent (B (word 'am name))) ...
  (method (n_usual n message) ...))

(define myC (instantiate C 'oski))
(ask myC 'n_usual 2 'name)
```

The last statement would return the name 'iamoski. Contrast this example with

```
(usual message)
```

which is not a method but an OOP construct because we do not need to use the expression

```
(ask class 'usual 'message)
```

Using the OOP constructs we already have available (e.g., **ask**, **parent**, **usual**, **initialize**, **default-method**, etc.) write a method **n_usual** that can be added to every class of an inheritance hierarchy to implement this semantics. **n_usual** should call the first method found that matches the message at that level. It only has to work for single inheritance (i.e., the solution we have in mind will not work for multiple inheritance).

Fill-in the following template:

```
(method (n_usual level message)
   (cond ((equal? level 0)
            (ask self message))
         ((equal? level 1)
            (ask usual message)                    )
-2       (else
            (ask usual 'n-usual (- level 1) message)))
```

6. (10 points) Answer the following questions about streams.

   (a) (2 points) The Scheme expression

   ```
   (define x (cons 1 x))
   ```

   when executed gives an error "unbound variable x." However, the following expression

   ```
   (define y (cons-stream 1 y))
   ```

   does not produce an error message. In fact, it defines an infinite stream of 1's. Explain briefly why the second expression does not cause an error while the first expression does.

   This occurs because cons-stream is a special form whose arguments are not evaluated. Y will only be evaluated when it is forced.

(b) (2 points) Given the following definitions for infinite streams:

```
(define (stream-map proc s1 s2)
  (cons-stream
    (proc (stream-car s1) (stream-car s2))
    (stream-map proc (stream-cdr s1) (stream-cdr s2))))
(define (add-streams s1 s2) (stream-map + s1 s2))

; define stream of 1's and positive integers
(define ones (cons-stream 1 ones))
(define positive-integers
  (cons-stream 1 (add-streams ones positive-integers)))
```

Describe the stream returned by:

```
(stream-map (lambda (a b) a) ones positive-integers)
```

the stream will be an infinite stream of ones

(c) (6 points) You are to create a stream of functions that includes the so-called CxR functions, that is, a stream with the functions

```
car
cdr
caar
cadr
cdar
cddr
...
```

Here is a template for creating this stream. You are to fill-in the blanks. Hint: you might find it convenient to have the first function in the stream return a CxR function with no *x* (i.e., the procedure returns the list passed to it as an argument).

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

— 6

```
(define cxrs
  (cons-stream
    (lambda (x) x)
    (interleave ((compose word 'ca) (xrs)
                ((compose word 'cd) (xrs))))
```

7