Wellesley College ◊ CS251 Programming Languages ◊ Spring 2000

PROBLEM SET 1
Due Friday, February 11

Reading: *SICP* 1.1—1.2; 2.1, 2.2—2.2.2, 2.3. Also read the Scheme tutorial, and the on-line documentation on Linux, Emacs, and MIT-Scheme.

Note: This is a long assignment. Please start early!

Problem 0: Linux Workstations

It is important to start learning how to use the Linux workstations. Although you can do some of the problems on this problem set without a computer, you should begin to familiarize yourself with the workstations as soon as possible so that you are comfortable with them when we begin to make heavier use of them. There is nothing to turn in for this problem.

a. Read the on-line documentation on Linux. This will walk you through the basics of logging onto a workstation and performing some common tasks in Unix.

b. Following the instructions in the on-line MIT-Scheme documentation, launch a Scheme interpreter in Linux and practice evaluating Scheme expressions. You will probably want to use this interpreter to check your answers on the other problems. (If you want to install a version of MIT-Scheme or some other version of Scheme on your personal computer, see the on-line documentation on Scheme implementations.)

c. Following the instructions in the on-line Emacs documentation, create an Emacs editing window and run the Emacs tutorial. This will walk you through using the most important features of Emacs. Use Emacs to create a file of Scheme definitions and expressions, and use

```
(load "<filename>")
```

in Scheme to load your file into the interpreter.

d. Returning to the on-line MIT-Scheme documentation, follow the instructions for starting Scheme within Emacs. Practice by evaluating some simple expressions and sending Emacs buffers to Scheme.

If you have any questions or problems while learning Linux, MIT Scheme, and Emacs, please do not hesitate to ask Lyn or Kirsten Chevalier for help!

Problem 1 [10]: Scheme Evaluation

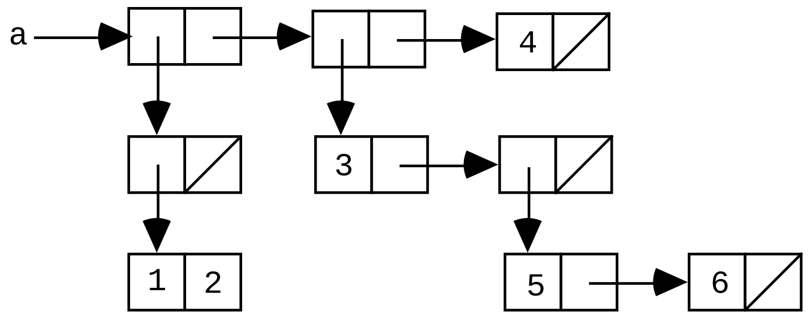
Give the result of evaluating the following Scheme expressions and definitions. Assume that the expressions are evaluated in order. If evaluating an expression gives an error, say so. You should figure out the answers without using the computer, but may use the Scheme interpreter to check your answers.

Note: Evaluating a definition does not return a value, but instead associates a name with a value. For each definition below, indicate the value that is associated with the name.

```
(define a 5)

(define b (* a a))

(+ (* 2 a) (- b a))

(2 * a)

(define average (lambda (x y) (/ (+ x y) 2)))

(average (* 2 a) (- b a))

(define c 'b)

(list a b c)

(list 'a 'b 'c)

(cons a b)

(cons a b c)

(a b c)

('a 'b 'c)

'(a b c)

(define apply-to-3-and-4 (lambda (f) (f 3 4)))

(apply-to-3-and-4 +)

(apply-to-3-and-4 *)

(apply-to-3-and-4 average)

(apply-to-3-and-4 (lambda (x y) x))

(apply-to-3-and-4 (if (> 1 2) + *))

(apply-to-3-and-4 (lambda (x y) (if (< x y) + *)))

(define add-a (lambda (x) (+ x a)))

(add-a 100)

(define a 17)

(add-a 100)

b

(define try (lambda (a) (add-a (* 2 a))))

(try 100)
```

Problem 2 [20]: Box-and-pointer diagrams

a. [15] Consider the following box-and-pointer diagram for the list structure named `a`:



i.[6] Using `car` and `cdr`, write Scheme expressions that extract the numbers 1 through 6 from `a`.

ii. [4] Write down the printed representation for `a`.

iii.[5] Give a Scheme expression that uses `cons` and `list` to create the structure depicted in the diagram.

You may wish to use a Scheme interpreter to check that your answers to i, ii, and iii are consistent.

b. [**5**] Consider the following Scheme definitions:

```
(define a (list '+ (* 2 3) '(- 3 4)))

(define b 'a)
```

Draw the box-and-pointer diagram corresponding to the value of the following expression:

```
(cons (list 'a a) (cons 'b b))
```

Problem 3 [20] This purpose of this problem is to encourage you to start thinking about comparing and evaluating languages. Below are a number of Scheme function definitions. For each part, either (1) translate the expression to a Java static method declaration that performs the same computation or (2) explain why such a translation cannot be performed. (Note: some of the answers could go either way depending on how "liberal" you are in you translations.)

a. `(define fun_a (lambda (x) (* (+ 1 2) (- 3 x)))`

b. `(define fun_b (lambda (x) (+ 1 2 3 x)))`

c. `(define fun_c (lambda (x) (if (= 1 2) (+ 3 x) (* 4 x))))`

d.
```
(define fun_d (lambda (x) (* (if (< x 10) (+ x 1) (* x 2))
                             (if (> x 20) (+ x 3) (* x 4)))))
```

e. `(define fun_e (lambda (x) ((if (< x 10) + *) (+ x 1) (* x 2))))`

f. `(define fun_f (lambda (*) (* 3 4)))`

g. `(define fun_g (lambda (f) (f 3 4))`

h. `(define fun_h (lambda (x y) (lambda (f) (f x y))))`

Problem 4 [50]: Recursion

Define the following Scheme procedures using recursion. For most of the problems it will be useful to define auxiliary procedures. Use auxiliary procedures defined in class if you find them helpful. Many of these problems are challenging; you are encouraged to work on them in groups.

a [5] `(sum-multiples-of-3-or-5 m n)`

Assume *m* and *n* are integers. Returns the sum of all integers from *m* up to *n* (inclusive) that are multiples of 5 and/or 5.

```
> (sum-multiples-of-3-or-5 0 10)
33  ; 3 + 5 + 6 + 9 + 10

> (sum-multiples-of-3-or-5 –9 12)
22

> (sum-multiples-of-3-or-5 18 18)
18

> (sum-multiples-of-3-or-5 10 0)
0 ; The range “10 up to 0” is empty.
```

b [5] `(all-contain-multiple? n intss)`

Assume that *n* is an integer and `intss is a list of lists of integers`. Returns `#t` if each list of integers in *intss* contains at least one integer that is a multiple of *n*; returns `#f` if some list of integers in *intss* does not contain a multiple of *n*. (Note that some Scheme interpreters use the empty list `()` to stand for `#f.)`

```
> (all-contain-multiple? 5 '((17 10 12) (25) (3 7 5)))
#t

> (all-contain-multiple? 3 '((17 10 12) (25) (3 7 5)))
#f

> (all-contain-multiple? 3 '())
#t
```

c [7] `(unzip lst)`
Assume that `lst` is a list of length *len* whose *i*th element is a list of the form (a$_i$ b$_i$). Return a list of the form `(lst1 lst2)` where `lst1` and `lst2` are length `len` lists whose *i*th elements are a$_i$ and b$_i$, respectively.

```
   > (unzip '((1 a) (2 b) (3 c)))
   ((1 2 3) (a b c))

   > (unzip '((1 a))
   ((1) (a))

   > (unzip '())
(() ())
```

d [7] `(cartesian-product` *`lst1 lst2`*`)`

Returns a list of all duples *`(a b)`* where *a* ranges over the elements of *`lst1`* and *b* ranges over the elements of *`lst2`*. The duples should be sorted first by the a entry (relative to the order in *`lst1`*) and then by the b entry (relative to the order in *`lst2`*).

```
> (cartesian-product '(1 2) '(a b c))
((1 a) (1 b) (1 c) (2 a) (2 b) (2 c))

> (cartesian-product '(2 1) '(c a b))
((2 c) (2 a) (2 b) (1 c) (1 a) (1 b))

> (cartesian-product '(c a b) ‘(2 1)
((c 2) (a 2) (b 2) (c 1) (a 1) (b 1))

> (cartesian-product '(1) '(a))
((1 a))

> (cartesian-product '() '(a b c))
()
```

e [8] `(count-atoms` *`sexp`*`)`
Return the number of atoms that appear in the s-expression *`sexp`* .

```
> (count-atoms '((a (b c)) d ((e f) g)))
7

> (count-atoms '(a b a b))
4

> (count-atoms 'a)
1

> (count-atoms '())
0
```

Your definition should have the following form:

```
(define count-atoms
  (lambda (sexp)
    (if (null? sexp)
        expression1
        (if (atom? sexp) expression2 expression3)))
```

f [8] `(deep-reverse` *`sexp`*`)`

Returns an s-expression whose elements are those of *`sexp`* reversed at every level.

```
   > (deep-reverse '((a (b c)) d ((e f) g)))
   ((g (f e)) d ((c b) a))

   > (deep-reverse '(a b c d))
   (d c b a)

   > (deep-reverse 'a)
   a

   > (deep-reverse '())
   ()
```

Your solution should have the form

```
   (define deep-reverse
     (lambda (sexp)
       (if (atom? sexp) expression1 expression2))
```

`where atom? is true of non-pairs:`

```
   (define (atom? val)
     (not (pair? val)))
```

You may find it helpful to use the `snoc` procedure in your definition (this is just `postpend` from CS111/CS230!)

```
   (define (snoc lst elt)
     (if (null? lst)
       (list elt)
       (cons (car lst) (snoc (cdr lst) elt))))
```

g [10] `(permutations` *`lst`*`)`

Assume that *`lst`* is a list of distinct elements (i.e., no duplicates). Returns a list of all the permutations of the elements of *`lst`*. The order of the permutations does not matter.

```
> (permutations ‘())
(())

> (permutations ‘(1))
((1))

> (permutations ‘(1 2))
((1 2) (2 1)) ; Order doesn’t matter

> (permutations ‘(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)) ; Order doesn’t matter
```

Extra Credit Problem [10]: Permutations in the presence of duplicates

This problem is optional. You should only attempt it after completing the rest of the problems.

Modify the permutations procedure from part g so that it correctly handles lists with duplicate elements. That is, each permutation of such a list should only be listed once in the result. You should *not* generate duplicate permutations and then remove them (e.g., by `remove-duplicates`). Rather, you should just not generate any duplicates to begin with.

`(permutations-dup` *`lst`*`)`
Return a list of all the permutations of the elements of *`lst`*.

```
> (permutations-dup ‘(2 1 2))
((1 2 2) (2 1 2) (2 2 1)) ; Order doesn’t matter

> (permutations ‘(a b a b b))
((a a b b b) (a b a b b) (a b b a b) (a b b b a)
 (b a a b b) (b a b a b) (b a b b a)
 (b b a a b) (b b a b a) (b b b a a)) ; Order doesn’t matter
```

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS251 Problem Set 1

Due Friday, February 11, 2000

Name:

Date & Time Submitted *(only if late)*:

Collaborators (*anyone you collaborated with in the process of doing the problem set)*:

*In the **Time** column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [10]		
Problem 2 [20]		
Problem 3 [20]		
Problem 4a [5]		
Problem 4b [5]		
Problem 4c [7]		
Problem 4d [7]		
Problem 4e [8]		
Problem 4f [8]		
Problem 4g [10]		
Extra Credit [10]		
Total		