CS251 Programming Languages **Handout #9**
Prof. Lyn Turbak **February 2, 2001**
Wellesley College

PROBLEM SET 1
Due Friday, February 9

Overview: The purpose of this assignment is to give you practice understanding and writing some simple Scheme programs. **This is a long assignment. It is strongly recommended that you (1) start early and (2) work with a partner. In particular you should plan to spend several days on the recursive list functions in Problem 3.**

Reading: *An Introduction to Scheme* (Handout #8); *SICP* 1.1—1.2; 2.1, 2.2—2.2.2, 2.3. Continue to study the documentation on Linux and Emacs (Handout #3) and MIT Scheme (Handout #7) as well as the new documentation on CVS (Handout #10).

Submission: Problems 1 and 2 are pencil-and-paper problems that only need to appear in your hardcopy submission. Problem 3 involves writing seven Scheme functions in the file `ps1-3.scm`. For this problem, your hardcopy submission should be your final version of `ps1-3.scm`. This file should also be your softcopy submission, which you should copy to the directory `~cs251/drop/ps1/`*`username`*, where *`username`* is your username.

Please attach to your hardcopy a problem set header sheet (which can be found at the end of this assignment) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment. If you work with a partner, you need only submit a single hardcopy and softcopy; please indicate on top of the problem set header sheet where the softcopy can be found.

Problem 1 [15]: Scheme Evaluation

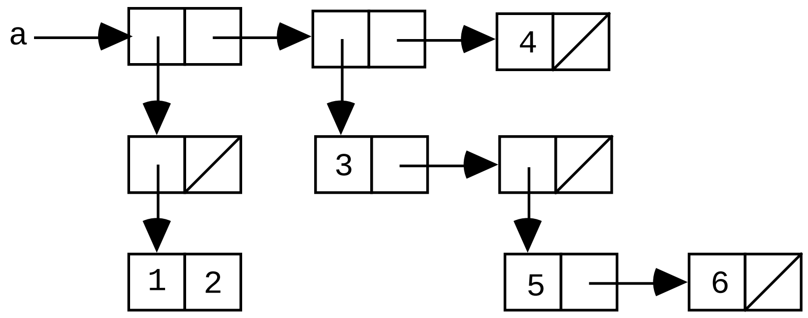
On the next page are a sequence of Scheme expressions and definitions. Show the details of how each expression is evaluated according to the substitution model and indicate the final results of each expression. Assume that the expressions are evaluated in order. If evaluating an expression gives an error, say so, and indicate what is the cause of the error. You should figure out the answers without using the computer, but may use a 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 [12]: Box-and-pointer diagrams

a. [10] Consider the following box-and-pointer diagram for the list structure named `a`:



i.[3] Using `car` and `cdr`, write Scheme expressions that extract the numbers 1 through 6 from `a`.

ii. [3] Write down the printed representation for `a`.

iii.[3] 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. [**3]** 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 [73]: List and Tree Recursion

This problem involves defining recursive functions that manipulate lists and trees in Scheme.

To do this problem, you will need to use several files that are in the CVS-controlled CS251 repository. Once you have set up your local CVS repository (see Handout #10 for details), you can get access to these files by executing the following in a Unix shell:

```
cd ~/cs251
cvs update -d
```

Indeed, every time you log in to a Linux machine to work on an assignment, you should execute the above commands to ensure that you have the most up-to-date versions of the problem set materials. Executing the above commands will create the local directory `~/cs251/ps1` containing two files:

- `ps1-3.scm`: This file contains skeletons for each of the procedures you are asked to define. You should flesh out each of the skeletons as you do the problems. In many of the problems it will also be helpful to define additional auxiliary procedures. You are welcome to use any procedures defined in class if you find them helpful.

- `ps1-3-test.scm`: This file contains code for testing each of your functions on some simple test cases. You can test the function in part *`ZZZ`* by loading this file and evaluating the Scheme form `(test-`*`ZZZ`*`)`. You can test all seven functions by evaluating `(test-all)`. Note that even if your function passes the test cases, it is not guaranteed to be correct; you are encouraged to extend the test cases in the testing file.

Below are the specifications for seven functions. Write definitions for each of the seven functions. Thinking carefully about your strategy before you start coding will save you lots of time! The divide-conquer-and-glue strategy you are familiar with from CS111 and CS230 can be use to solve all problems.

a [6] `(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 3 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 [7] `(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 [10] `(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 [10] `(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) (c 1) (a 2) (a 1) (b 2) (b 1))

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

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

e [10] `(count-atoms sexp)`
Return the number of atoms (non-pairs) 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)))
```

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

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

f [10] `(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 (null? sexp)
        expression1
        (if (atom? sexp) expression2 expression3))))
```

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 [20] `(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

> (permutations ‘(1 2 3 4))
((1 2 3 4) (1 2 4 3) (1 3 2 4) (1 3 4 2) (1 4 2 3) (1 4 3 2)
 (2 1 3 4) (2 1 4 3) (2 3 1 4) (2 3 4 1) (2 4 1 3) (2 4 3 1)
 (3 1 2 4) (3 1 4 2) (3 2 1 4) (3 2 4 1) (3 4 1 2) (3 4 2 1)
 (4 1 2 3) (4 1 3 2) (4 2 1 3) (4 2 3 1) (4 3 1 2) (4 3 2 1))
; Order doesn’t matter
```

Extra Credit Problem [20]: 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-dup ‘(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 9, 2001

Names of Team Members:

Date & Time Submitted:

Soft Copy Directory:

Collaborators (*any teams collaborated with in the process of doing the problem set)*:

In the ***Time*** *column, please estimate the total time each team member 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 For** ______ **(Team Member #1)**	**Time For** ______ **(Team Member #2)**	**Score**
General Reading			
Problem 1 [15]			
Problem 2 [15]			
Problem 3a [5]			
Problem 3b [5]			
Problem 3c [10]			
Problem 3d [10]			
Problem 3e [10]			
Problem 3f [10]			
Problem 3g [20]			
Extra Credit [20]			
Total			