

University of Massachusetts Lowell

91.301: Organization of Programming Languages
Fall 2004

Problem Set 3 Lists

Issued: Section 201 (Dimock): Friday, 24 Sept 2004; Section 202 (Yanco): Thursday, 23 Sept 2004
Due: Section 201: Friday, 1 Oct 2004; Section 202: Thursday, 30 Sept 2004
Reading: Text (SICP 2nd Edition by Abelson & Sussman): Section 2.2

Overview

In this problem set, you will write procedures that manipulate lists of numbers.

What to turn in

Keep your answers in a file named `ps3-ans.ss` (the file that saves your definitions buffer). Put the code for each problem sequentially in the definitions buffer. (If you work problems in a different sequence, put them in order before you submit.) Put the problem number in a comment before the code for that problem (use a semi-colon to make a line a comment). Underneath the code for each problem, cut and paste the appropriate sample runs. Please put semi-colons in front of the sample runs, which will comment them out and allow you to load the entire buffer in another session.

Print the definitions buffer (or the file `ps3-ans.ss`, which should be equivalent) and turn it in during the class on the due date. Also submit the answers using the `submit` command:

Section 201: `submit dimock ps3 ps3-ans.ss`

Section 202: `submit holly 301-ps3 ps3-ans.ss`

Problems

Problem 1 Draw the box and pointer diagrams resulting from the following groups of expressions (one diagram for each group). You may turn these in hard copy only, rather than trying to create an ASCII or drawing program representation. (All of the rest of the exercises should be turned in as hard and soft copy.)

a) `(define a (cons 1 (cons 2 3)))`

`(define b (cons 4 a))`

`(define c (list a b))`

b) `(define d (list 5 6 7))`

`(define e (list 8 d 9))`

Problem 2: Consider the following recursive procedure:

```
(define (list-sum lst)
  (if (null? lst)
      0
      (+ (car lst)
          (list-sum (cdr lst)))))
```

- How many times is `list-sum` called when evaluating the expression `(list-sum '(1 2 3))` ?
- What is the order of growth in space and time for the `list-sum` procedure above?
- Write an iterative version of the procedure `list-sum`.
- What is the order of growth in space and time for your iterative `list-sum` procedure from part c?

Problem 3: Write a definition of `list-product`, which resembles the `list-sum` procedure above, but returns the product of the elements of the list instead.

Problem 4: Write a definition of a procedure named `list-average`, which takes a list of numbers as its argument, and returns the average of the numbers in the list.

Problem 5: Write a procedure `pair-or-null?` which takes an argument named `thing` and returns a boolean saying whether the thing is either a pair or is the empty list. Use the predicates `pair?` and `null?` to check for these.

Problem 6: Write a procedure called `number-list?` which checks if its argument is a list of numbers. It should take one argument and check whether it conforms to the following recursive definition of a list of numbers:

A thing is a list of numbers if it's either

- the empty list, or
- a pair whose `car` is a number and whose `cdr` is a list of numbers

Problem 7: Write a procedure called `list-ascending?`, which tests if a list of one or more numbers is in ascending (increasing) order. It should take one argument, which you can assume is a list of numbers.

A list of numbers is ascending if

- it's an empty list
- it's a one-element list
- it's a list of two or more elements, where the first element is less than or equal to the second, and the rest of the list (starting at the second element) is an ascending list.

Problem 8: Recall the built-in Scheme procedure `append`, which appends two lists and creates a single list with all of their elements.

For example, `(append '(1 2 3) '(4 5 6))` returns `(1 2 3 4 5 6)`.

A common mistake is to attempt to add one thing to the end of a list using `append`. This doesn't work, because `append` expects all of its arguments to be lists.

For example, if you try to `append 'baz` to the end of `'(foo bar)`, you don't get `(foo bar baz)`, you get `(foo bar . baz)`.

The right way to `append` one item to the end of the list is to make a one-element list holding that item, and then `append` the two lists.

Write a procedure named `snoc` which takes two arguments, which are a list and an item to be `snoc'd` onto the end of the list.

`snoc` should return a new list containing all of the elements of the original list, followed by the new item. (We call this procedure `snoc`, because it's like `cons`, only backwards—it takes one element and one list, and returns one list.)