

Problem Set 3

Out: Thursday, 9 February 2006
Due: Thursday, 16 February 2006
Reading: Sections 2.1 and 2.2

Overview

This problem set covers data abstractions and lists as data structures.

Scheme Settings

- Use the Standard (R5RS) language in DrScheme
- You should define nil as follows in your evaluation buffer:

```
(define nil '())
```

What to turn in

Keep your answers in a file named ps3-ans.scm. Put the code for each problem sequentially in the definitions buffer. 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. You can 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 answers buffer and turn it in at the beginning of class on the due date. Staple any handwritten portions of the assignment to your code file that you turn in. Also submit your answer buffer using the submit command on mercury:

```
submit holly 301-ps3 ps3-ans.scm
```

Exercises

Problem 1: In class, we initially defined `make-rat` as follows:

```
(define (make-rat n d)
  (cons n d))
```

We then rewrote the code to simplify for the greatest common divisor:

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/d g))))
```

However, this version doesn't handle negative numbers well. Redefine `make-rat` so that if the fraction is negative, the negative sign is attached to the numerator, not the denominator.

Problem 2: For each of the following definitions,

- a. Draw the box and pointer diagram
- b. Give the sequence of cars and cdrs needed to return the number 2

```
(define a (list 1 2 3 4 5))
```

```
(define b (cons (cons 1 2) (cons 3 4)))
```

```
(define c (cons 1 (list 2 3)))
```

Problem 3: Draw a box and pointer diagram for each of the lists d, e and f defined as follows:

```
(define d (list 1 2 3 4))
```

```
(define e (cons d d))
```

```
(define f (append d d))
```

Problem 4: Consider the following recursive procedure:

```
(define (list-prod lst)
  (if (null? lst)
      1
      (* (car lst)
         (list-prod (cdr lst)))))
```

- a) How many times is list-prod called when evaluating the expression (list-prod '(1 2 3 4))?
- b) What is the order of growth in space and time for the list-prod procedure above?
- c) Write an iterative version of the procedure list-prod.
- d) What is the order of growth in space and time for your iterative list-prod procedure from part c?

Problem 5: Write a procedure called last-element which takes a list as its argument and returns the last element of the list. For example,

```
(last-element (list 1 2 3 4))
```

should return

```
4
```

(Be sure that you're returning the single element, not a list of that element.)

Problem 6: Write a procedure called `triple-list` in three different ways:

- a. Write a list manipulation procedure of your own to do this.
- b. Use the `scale-list` procedure on p. 105 in your definition.
- c. Use the `map` procedure on p. 105 in your definition.

Problem 7: Exercise 2.21 on p. 106.

Problem 8: Exercise 2.33 on p. 119.