CS 1 Fall 2008

Lab 5: The Little Evaluator That `cdr`'ed

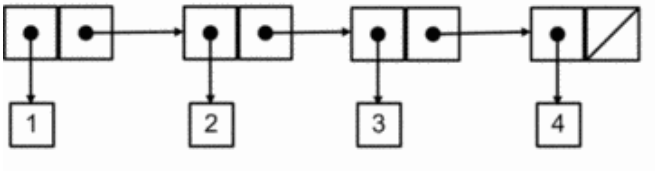
Assigned: ***Monday, October 27, 2008***
Due: ***Thursday, November 6, 2008, 02:00:00***

You should read through the end of Section 2.2 in order to be fully prepared for this lab. This lab is shorter than usual to give you time to do the midterm.

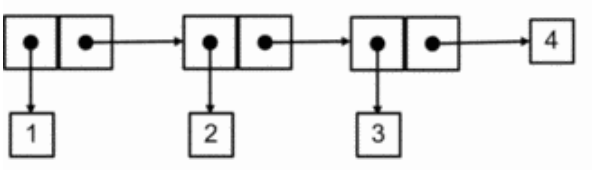
Part A: Exercises

Please complete the following exercises before your recitation on Friday. Should you have any difficulties working these basic exercises, bring your questions to recitation.

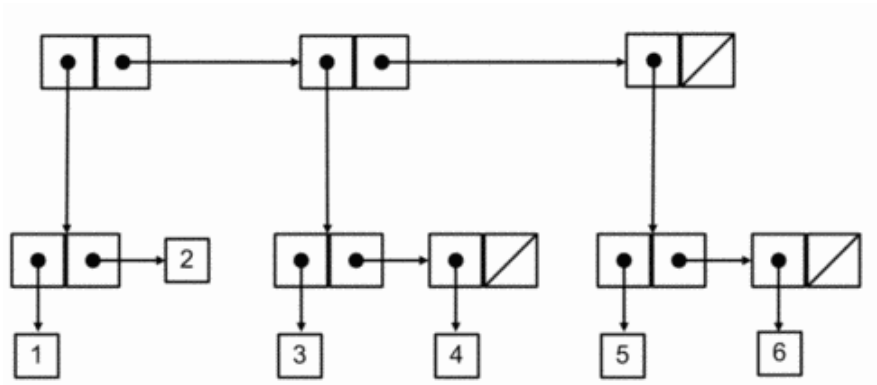
1. **[25]** Give a Scheme expression which will build each of the structures shown in the box and pointer diagrams:

1. 

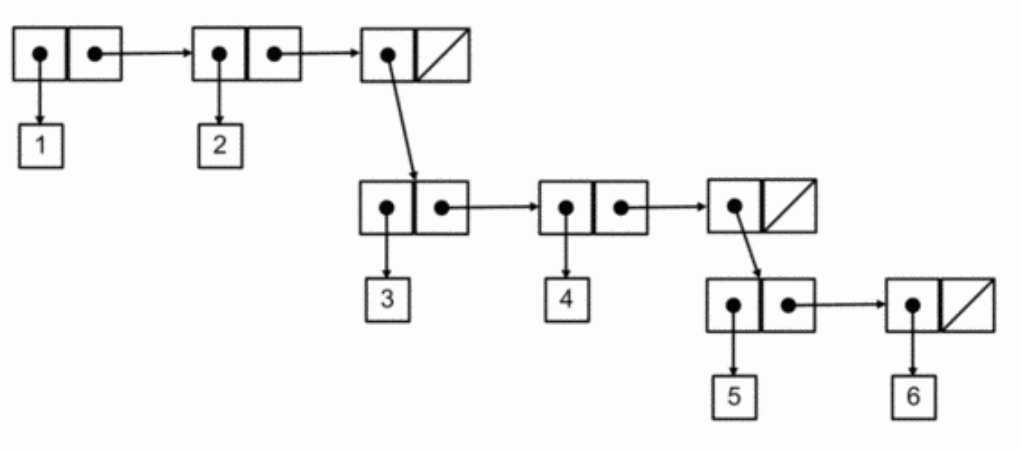
---

2. 

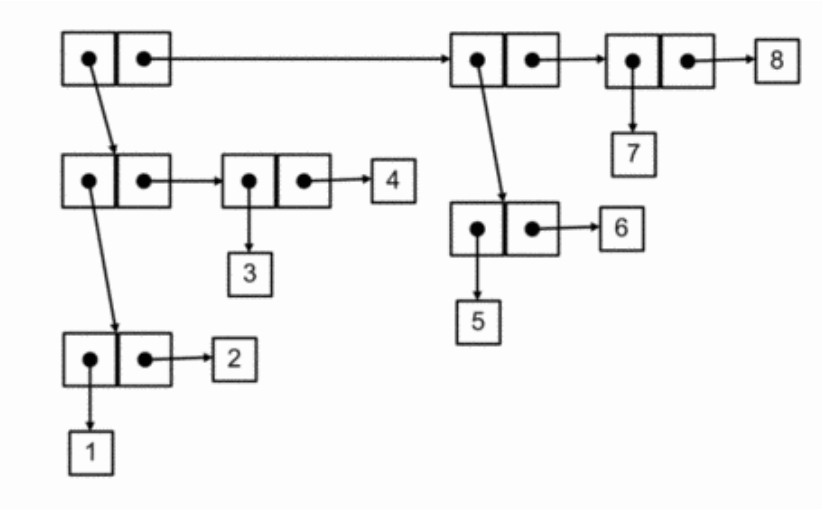
---

3. 

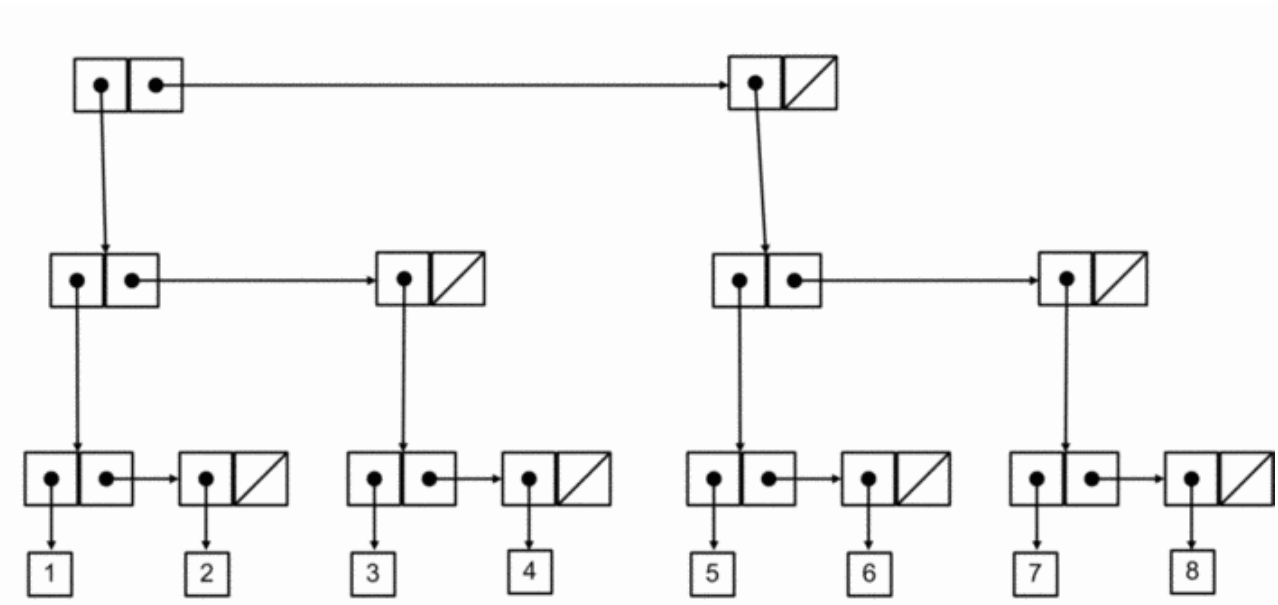
4.



5.



6.



---

2. **[15]** What's wrong with these expressions? Describe why each of them will result in an error.

   a. `(cdr (list))`

   b. `(cdr (cdr (cdr (list 1 2))))`

   c. `(car (car (list 1 2)))`

d.
```
(define (buggy-sum lst)
  (+ (car lst) (buggy-sum (cdr lst))))
(buggy-sum (list 1 2 3))
```

e.
```
(+ 1 2 3 (list 1 2 3))
```

After figuring out what's wrong with them, type them into the Scheme interpreter and see how the interpreter will respond to these "bugs".

Part B: Working with lists

1. **[15]** Explain the function of the following procedure which operates on lists:

```
(define (fh x)
  (define (fh-aux l aux)
     (cond ((null? l) (list))
           ((null? (cdr l)) (list))
           (else (cons (car aux) (fh-aux (cdr (cdr l)) (cdr aux))))))
  (fh-aux x x))
```

2. Write the following procedures which involve lists. Include a contract and comment in all your functions, as well as some tests using `check-expect`.

   a. **[15]** A procedure that counts the negative elements in a list and returns the count.

   b. **[10]** A procedure that takes in an integer, n, and creates a list containing the first n powers of 2 starting with 2^0=1.

   c. **[25]** A procedure that takes in a list of numbers, and returns a list containing the prefix sum of the original list. *e.g.* `(prefix-sum (list 1 3 5 2)) ==> (1 4 9 11)`. The prefix sum is the sum of all of the elements in the list up to that point, so for the list `(1 3 5 2)` the prefix sum is `(1, 1+3, 1+3+5, 1+3+5+2)` or `(1 4 9 11)`.

Part C: Book Exercises

Please do the following exercises from the textbook. We've added some clarifications which should make the problems easier to manage. Add contracts, comments, and tests to all the functions you write (but not to the functions defined in the book or included below in their entirety).

1. **[30]** Exercise 2.21 (using `map`). Read the section just preceding the exercise to learn how the `map` higher-order procedure works. `map` is actually one of the most frequently-used higher-order procedures. Use `(list)` where the book says to use `nil` (coding standards have changed since the book was written).

2. **[30]** Exercise 2.32 (generate all subsets of a set, fill in code and explain). Again, use `(list)` where the book uses `nil`.

   Note that the `append` procedure concatenates two lists, returning a new list containing all the elements of both old lists in order. So, for instance, `(append (list 1 2 3 4) (list 5 6 7 8))` gives the result `(1 2 3 4 5 6 7 8)` without altering either of the first two lists.

   Hint: The code you need to add is very short; it will easily fit on the same line as the `map`.

   Be careful with the contract here; make it as general as possible. Also note that the tests may fail because the order is wrong; if this happens you can reorder the values in the test to make it work.

3. **[30]** Exercise 2.34 (evaluate polynomials).

Note that the `accumulate` procedure is defined as:

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

and is discussed at length in section 2.2.3 of SICP. (You don't need to write contracts/comments/tests for `accumulate`.)

To solve this problem, it may be helpful to take an example (*e.g.* `(horner-eval 2 (list 1 3 0 5 0 1))`, figure out what computation you want to be performed, and then write the code so that it performs it. Note that the coefficients in the list are in reverse order, so the first 1 is the x^0 coefficient and the last 1 is the x^5 coefficient. Again, the amount of code you have to add here is extremely small. *Hint:* Think carefully about what the arguments of the lambda expression refer to.

To save you some time, here are some tests you can use:

```
(check-expect (horner-eval 0 (list)) 0)
(check-expect (horner-eval 1 (list)) 0)
(check-expect (horner-eval 0 (list 42)) 42)
(check-expect (horner-eval 1 (list 42)) 42)
(check-expect (horner-eval 0 (list 1 2 3)) 1)
(check-expect (horner-eval 1 (list 1 2 3)) 6)
(check-expect (horner-eval 2 (list 1 2 3)) 17)
```

4. **[45]** Exercise 2.37 (matrix calculations; dot product, transpose). Note that you'll need to read exercise Exercise 2.36 to familiarize yourself with `accumulate-n`. A working definition of `accumulate-n` is:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))  ; sequences are empty
      (list)  ; we prefer this to nil, since nil isn't predefined in DrScheme
      (cons (accumulate   op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))
```

None of the code fragments you need to fill in in any of the functions in this exercise are more than a line long. Feel free to use the earlier functions in the definitions of the later functions. You don't need to write contracts/comments/tests for `accumulate` or `accumulate-n`.

Part D: Good Code, Bad Code

There is no part D in this week's lab (and there was much rejoicing).

[end lab 5]