Exam Prep 7: Scheme

Students from past semesters wanted more content and structured time to prepare for exams. Exam Prep sections are a way to solidify your understanding of the week's materials. The problems are typically designed to be a bridge between discussion/lab/homework difficulty and exam difficulty.

Reminder: There is nothing to turn in and there is no credit given for attending Exam Prep Sections.

We try to make these problems **exam level** , so you are not expected to be able to solve them coming straight from lecture without additional practice. To get the most out of Exam Prep, we recommend you **try these problems first on your own** before coming to the Exam Prep section, where we will explain how to solve these problems while giving tips and advice for the exam. Do not worry if you struggle with these problems, **it is okay to struggle while learning**.

You can work with anyone you want, including sharing solutions. We just ask you don't spoil the problems for anyone else in the class. Thanks!

You may only put code where there are underscores for the codewriting questions.

You can test your functions on their doctests by clicking the red test tube in the top right corner after clicking Run in 61A Code. Passing the doctests is not necessarily enough to get the problem fully correct. You must fully solve the stated problem.

Q1: Slice It!

Difficulty: ⭐⭐

Implement the `get-slicer` procedure, which takes integers `a` and `b` and returns an *a-b slicing function*. An `a-b` slicing function takes in a list as input and outputs a new list with the values of the original list from index `a` (inclusive) to index `b` (exclusive).

Your implementation should behave like Python slicing, but should assume a step size of one with no negative slicing indices. Indices start at zero.

Note: the skeleton code is just a suggestion. Feel free to use your own structure if you prefer.

```
(define (get-slicer a b)
  (define (slicer lst)
    (define (slicer-helper c i j)
      (cond
        ((or (null? c) (>= i j)) nil)
        ((= i 0) (cons (car c) (slicer-helper (cdr c) i (- j 1))))
        (else (slicer-helper (cdr c) (- i 1) (- j 1)))))
    (slicer-helper lst a b))
  slicer)

; DOCTESTS (No need to modify)
(define a '(0 1 2 3 4 5 6))
(define one-two-three (get-slicer 1 4))
(define one-end (get-slicer 1 10))
(define zero (get-slicer 0 1))
(define empty (get-slicer 4 4))

(expect (one-two-three a) (1 2 3))
(expect (one-end a) (1 2 3 4 5 6))
(expect (zero a) (0))
(expect (empty a) ())

```

Q2: Partition Options

Difficulty: ⭐⭐

First, write the procedure `combine`, which takes in lists `lst1` and `lst2`, as well as a value `x`, and outputs a new list with `x` before each item of `lst1`, and then all elements of `lst2` unmodified. For examples, see the doctests.

Then write the procedure `partition-options`, which takes in positive integers `total` and `biggest` and outputs a list of lists, in which each inner list contains numbers no larger than `biggest` that sum to `total`.

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
; helper function
(define (combine lst1 lst2 x)
  (if (null? lst1)
      lst2
      (cons (cons x (car lst1))
            (combine (cdr lst1) lst2 x))))

(expect (combine '((1)) '((3) (4 5)) 0)
        ((0 1) (3) (4 5)))

(expect (combine '((1) (2 3) (3)) '((4 5) (6)) 'a)
        ((a 1) (a 2 3) (a 3) (4 5) (6)))

(define (partition-options total biggest)
  (cond
    ((= 0 total)
     '(()))
    ((or (< total 0) (= 0 biggest))
     '())
    (else
     (let ((with
            (partition-options (- total biggest) biggest))
```

```
22                (partition-options (- total biggest) biggest))
```

Q3: Find It!

Difficulty: ☆☆

Implement the `find-in-tree` procedure, which takes a Scheme tree `t`, a target value `goal` and returns a list containing the node values on a path from the root of `t` to a node containing `goal`. If no such path exists, `find-in-tree` returns an empty list.

Complete `find-in-branches` for ease of implementation. `find-in-branches` takes a list of branches `bs` and a value `goal` and returns a list containing the node values on a path from any branch to a node containing `goal`. If no such path exists, `find-in-branches` returns an empty list.

The Scheme tree ADT is specified at the top of the skeleton code.

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
(define (tree label branches) (cons label branches))
(define (label t) (car t))
(define (branches t) (cdr t))
(define (is-leaf t) (null? (branches t)))

(define (find-in-tree t goal)
  (if (eq? (label t) goal)
    (list goal)
    (let ((path (find-in-branches (branches t) goal)))
      (if (null? path)
        nil
        (cons (label t) path)))))

(define (find-in-branches bs goal)
  (if (null? bs)
    nil
    (let ((path (find-in-tree (car bs) goal)))
      (if (null? path)
        (find-in-branches (cdr bs) goal)
```

```
          (find-in-branches (car tr) goal)
          path))))

; DOCTESTS (no need to modify)
(define t1 (tree 1
  (list
    (tree 2
      (list
        (tree 5 nil)
          (tree 6 (list
            (tree 8 nil)))))
    (tree 3 nil)
    (tree 4
      (list
        (tree 7 nil))))))

(expect (find-in-tree t1 7) (1 4 7))
(expect (find-in-tree t1 1) (1))
(expect (find-in-tree t1 12) ())
```