REVIEW: SCHEME, RECURSION, AND GENERATORS

CSM 61A

April 13, 2020 - April 15, 2020

1 Let in Scheme

let is a special form in Scheme which allows you to create local bindings. Consider the following example:

```
(let ((x 1)) (+ x 1))
```

Here, we assign `x` to `1`, and then evaluate the expression `(+ x 1)` using that binding, returning `2`. However, outside of this expression, `x` would not be bound to anything.
Each `let` special form has a corresponding lambda equivalent. The equivalent lambda expression for the above example is:

```
((lambda (x) (+ x 1)) 1)
```

1. The following line of code does not work. Why? Write the lambda equivalent of the `let` expressions.

```
(let ((foo 3)
      (bar (+ foo 2)))
    (+ foo bar))
```

2 Scheme

1. Suppose Isabelle bought turnips from the Stalk Market and has stored them in random amounts among an ordered sequence of boxes. By the magic of time travel, Isabelle's friend Tom Nook can fast-forward one week into the future and determine exactly how many of Isabelle's turnips will rot over the week and have to be discarded.

   Assuming that boxes of turnips will rot in order, i.e. all of box 1's turnips will rot before any of box 2's turnips, help Isabelle determine which turnips will still be fresh by week's end. Specifically, fill in `decay`, which takes in a list of positive integers `boxes`, which represents how many turnips are in each box, and a positive integer `rotten` representing the number of turnips that will rot, and returns a list of non-negative integers that represents how many fresh turnips will remain in each box.

```
; doctests
scm> (define a '(1 6 3 4))
a
scm> (decay a 1)
(0 6 3 4)
scm> (decay a 5)
(0 2 3 4)
scm> (decay a 9)
(0 0 1 4)
scm> (decay a 1000)
(0 0 0 0)

(define (decay boxes rotten)









)
```

2. Fill in `backwards-`**`sum`** such that it takes in a list of numbers `lst` and returns a new list with each element being the sum of itself and all elements to the right of it in `lst`.

   Sidebar: the word "sum" being bolded has no significance, it is an auto-formatting issue.

```
; doctests
scm> (backwards-sum '(1 2 3 4))
(10 9 7 4)
scm> (backwards-sum '(2 -1 3 7))
(11 9 10 7)

(define (backwards-sum lst)




)
```

3 Recursion

1. Fill in `collapse`, which takes in a non-negative integer `n` and returns the number resulting from removing all digits that are equal to an adjacent digit, i.e. the number has no adjacent digits that are the same.

```
def collapse(n):
    """
    >>> collapse(12234441)
    12341
    >>> collapse(11200000013333)
    12013
    """
    rest, last = n // 10, n % 10

    if _____________________________________:

        _____________________________________

    elif _____________________________________:

        _____________________________________
    else:

        _____________________________________
```

2. Fill in `combine_to_61`, which takes in a list of positive integers and returns `True` if a contiguous sublist (i.e. a sublist of adjacent elements) combine to 61. You can **combine** two adjacent elements by either summing them or multiplying them together. If there is no combination of summing and multiplying that equals 61, return `False`.

```
def combine_to_61(lst):
    """
    >>> combine_to_61([3, 4, 5])
    False # no combination will produce 61
    >>> combine_to_61([2, 6, 10, 1, 3])
    True # 61 = 6 * 10 + 1
    >>> combine_to_61([2, 6, 3, 10, 1])
    False # elements must be contiguous
    """

    def helper(lst, num_so_far):

        if _____________________________________:
            return True

        elif _____________________________________:
            return False

        with_sum = _____________________________ and \

            helper(_____________________, _____________________)

        with_mul = _____________________________ and \

            helper(_____________________, _____________________)
        return with_sum or with_mul

    return _____________________________________
```

3. Implement the function `make_change`, which takes in a non-negative integer amount in cents `n` and returns the minimum number of coins needed to make change for `n` using 1-cent, 3-cent, and 4-cent coins.

```
def make_change(n):
    """
    >>> make_change(5) # 5 = 4 + 1 (not 3 + 1 + 1)
    2
    >>> make_change(6) # 6 = 3 + 3 (not 4 + 1 + 1)
    2
    """

    if _____________________:
        return 0

    elif _____________________:

        ___________________________________________

    elif _____________________:

        ___________________________________________
    else:

        ___________________________________________
```

4 Generators

1. Define a generator function `in_order`, which takes in a tree `t`, and assume that `t` has either 0 or 2 branches only. Fill in `in_order` so that it returns a generator that yields the labels of `t` in the following order: first branch node, parent node, second branch node.

```
def in_order(t):
    """
    >>> t = Tree(0, [Tree(1), Tree(2, [Tree(3), Tree(4)])])
    >>> list(in_order(t))
    [1, 0, 3, 2, 4]
    """

    if _____________________:

        yield _____________________________
    else:

        lst = _____________________________

        for _____________________________________:

            _____________________________________
```