Lab 13: Final Review **lab13.zip (lab13.zip)**

Due by 11:59pm on Tuesday, August 10.

Starter Files

Download lab13.zip (lab13.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

The questions in this assignment are not graded, but they are highly recommended to help you prepare for the upcoming final. You will receive credit for this lab even if you do not complete these questions.

Suggested Questions

Trees

Q1: Prune Min

Write a function that prunes a `Tree t` mutatively. `t` and its branches always have zero or two branches. For the trees with two branches, reduce the number of branches from two to one by keeping the branch that has the smaller label value. Do nothing with trees with zero branches.

Prune the tree in a direction of your choosing (top down or bottom up). The result should be a linear tree.

```
def prune_min(t):
    """Prune the tree mutatively from the bottom up.

    >>> t1 = Tree(6)
    >>> prune_min(t1)
    >>> t1
    Tree(6)
    >>> t2 = Tree(6, [Tree(3), Tree(4)])
    >>> prune_min(t2)
    >>> t2
    Tree(6, [Tree(3)])
    >>> t3 = Tree(6, [Tree(3, [Tree(1), Tree(2)]), Tree(5, [Tree(3), Tree(4)])])
    >>> prune_min(t3)
    >>> t3
    Tree(6, [Tree(3, [Tree(1)])])
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q prune_min
```

Q2: Add trees

Define the function `add_trees`, which takes in two trees and returns a new tree where each corresponding node from the first tree is added with the node from the second tree. If a node at any particular position is present in one tree but not the other, it should be present in the new tree as well. At each level of the tree, nodes correspond to each other starting from the leftmost node.

> *Hint*: You may want to use the built-in zip function to iterate over multiple sequences at once.
>
> *Note*: If you feel that this one's a lot harder than the previous tree problems, that's totally fine! This is a pretty difficult problem, but you can do it! Talk about it with other students, and come back to it if you need to.

```
def add_trees(t1, t2):
    """
    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(add_trees(numbers, numbers))
    2
      4
        6
        8
      10
        12
          14
        16
    >>> print_tree(add_trees(tree(2), tree(3, [tree(4), tree(5)])))
    5
      4
      5
    >>> print_tree(add_trees(tree(2, [tree(3)]), tree(2, [tree(3), tree(4)])))
    4
      6
      4
    >>> print_tree(add_trees(tree(2, [tree(3, [tree(4), tree(5)])]), \
    tree(2, [tree(3, [tree(4)]), tree(5)])))
    4
      6
        8
        5
      5
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q add_trees
```

Scheme

Q3: Split

Implement `split-at`, which takes a list `lst` and a non-negative number `n` as input and returns a pair `new` such that `(car new)` is the first `n` elements of `lst` and `(cdr new)` is the remaining elements of `lst`. If `n` is greater than the length of `lst`, `(car new)` should be `lst` and `(cdr new)` should be `nil`.

```
    scm> (car (split-at '(2 4 6 8 10) 3))
    (2 4 6)
    scm> (cdr (split-at '(2 4 6 8 10) 3))
    (8 10)
```

```
(define (split-at lst n)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q split-at
```

Q4: Compose All

Implement `compose-all`, which takes a list of one-argument functions and returns a one-argument function that applies each function in that list in turn to its argument. For example, if `func` is the result of calling `compose-all` on a list of functions `(f g h)`, then `(func x)` should be equivalent to the result of calling `(h (g (f x)))`.

```
scm> (define (square x) (* x x))
square
scm> (define (add-one x) (+ x 1))
add-one
scm> (define (double x) (* x 2))
double
scm> (define composed (compose-all (list double square add-one)))
composed
scm> (composed 1)
5
scm> (composed 2)
17
```

```
(define (compose-all funcs)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q compose-all
```

Tree Recursion

Q5: Align Skeleton

Have you wondered how your CS61A exams are graded online? To see how your submission differs from the solution skeleton code, `okpy` uses an algorithm very similar to the one below which shows us the minimum number of edit operations needed to transform the the skeleton code into your submission.

Similar to `pawssible_patches` in Cats, we consider two different edit operations:

1. Insert a letter to the skeleton code
2. Delete a letter from the skeleton code

Given two strings, `skeleton` and `code`, implement `align_skeleton`, a function that minimizes the edit distance between the two strings and returns a string of all the edits. Each addition is represented with `+[]`, and each deletion is represented with `-[]`. For example:

```
>>> align_skeleton(skeleton = "x=5", code = "x=6")
'x=+[6]-[5]'
>>> align_skeleton(skeleton = "while x<y", code = "for x<y")
'+[f]+[o]+[r]-[w]-[h]-[i]-[l]-[e]x<y'
```

In the first example, the `+[6]` represents adding a "6" to the skeleton code, while the `-[5]` represents removing a "5" to the skeleton code. In the second example, we add in the letters "f", "o", and "r" and remove the letters "w", "h", "i", "l", and "e" from the skeleton code to transform it to the submitted code.

> Note: For simplicity, all whitespaces are stripped from both the skeleton and submitted code, so you don't have to consider whitespaces in your logic.

`align_skeleton` uses a recursive helper function, `helper_align`, which takes in `skeleton_idx` and `code_idx`, the indices of the letters from `skeleton` and `code` which we are comparing. It returns two things: match, the sequence of edit corrections, and cost, the numer of edit operations made. First, you should define your three base cases:

- If both `skeleton_idx` and `code_idx` are at the end of their respective strings, then there are no more operations to be made.
- If we have not finished considering all letters in `skeleton` but we have considered all letters in `code`, then we simply need to delete all the remaining letters in `skeleton` to match it to `code`.
- If we have not finished considering all letters in `code` but we have considered all letters in `skeleton`, then we simply need to add all the remaining letters in `code` to `skeleton`.

Next, you should implement the rest of the edit operations for `align_skeleton` and `helper_align`. You may not need all the lines provided.

```
def align_skeleton(skeleton, code):
    """
    Aligns the given skeleton with the given code, minimizing the edit distance betw
    the two. Both skeleton and code are assumed to be valid one-line strings of code

    >>> align_skeleton(skeleton="", code="")
    ''
    >>> align_skeleton(skeleton="", code="i")
    '+[i]'
    >>> align_skeleton(skeleton="i", code="")
    '-[i]'
    >>> align_skeleton(skeleton="i", code="i")
    'i'
    >>> align_skeleton(skeleton="i", code="j")
    '+[j]-[i]'
    >>> align_skeleton(skeleton="x=5", code="x=6")
    'x=+[6]-[5]'
    >>> align_skeleton(skeleton="return x", code="return x+1")
    'returnx+[+]+[1]'
    >>> align_skeleton(skeleton="while x<y", code="for x<y")
    '+[f]+[o]+[r]-[w]-[h]-[i]-[l]-[e]x<y'
    >>> align_skeleton(skeleton="def f(x):", code="def g(x):")
    'def+[g]-[f](x):'
    """
    skeleton, code = skeleton.replace(" ", ""), code.replace(" ", "")

    def helper_align(skeleton_idx, code_idx):
        """
        Aligns the given skeletal segment with the code.
        Returns (match, cost)
            match: the sequence of corrections as a string
            cost: the cost of the corrections, in edits
        """
        if skeleton_idx == len(skeleton) and code_idx == len(code):
            return _________, _______________
        if skeleton_idx < len(skeleton) and code_idx == len(code):
            edits = "".join(["-[" + c + "]" for c in skeleton[skeleton_idx:]])
            return _________, _______________
        if skeleton_idx == len(skeleton) and code_idx < len(code):
            edits = "".join(["+[" + c + "]" for c in code[code_idx:]])
            return _________, _______________

        possibilities = []
        skel_char, code_char = skeleton[skeleton_idx], code[code_idx]
        # Match
        if skel_char == code_char:
            ________________________________________
```

```
            ____________________________________
            possibilities.append((_______, ______))
        # Insert
        ____________________________________
        ____________________________________
        possibilities.append((_______, ______))
        # Delete
        ____________________________________
        ____________________________________
        possibilities.append((_______, ______))
        return min(possibilities, key=lambda x: x[1])
    result, cost = _________________________
    return result
```

Use Ok to test your code:

```
python3 ok -q align_skeleton
```

OOP

Q6: Keyboard

We'd like to create a `Keyboard` class that takes in an arbitrary number of `Button`s and stores these `Button`s in a dictionary. The keys in the dictionary will be ints that represent the postition on the `Keyboard`, and the values will be the respective `Button`. Fill out the methods in the `Keyboard` class according to each description, using the doctests as a reference for the behavior of a `Keyboard`.

```
class Button:
    """
    Represents a single button
    """
    def __init__(self, pos, key):
        """
        Creates a button
        """
        self.pos = pos
        self.key = key
        self.times_pressed = 0

class Keyboard:
    """A Keyboard takes in an arbitrary amount of buttons, and has a
    dictionary of positions as keys, and values as Buttons.

    >>> b1 = Button(0, "H")
    >>> b2 = Button(1, "I")
    >>> k = Keyboard(b1, b2)
    >>> k.buttons[0].key
    'H'
    >>> k.press(1)
    'I'
    >>> k.press(2) #No button at this position
    ''
    >>> k.typing([0, 1])
    'HI'
    >>> k.typing([1, 0])
    'IH'
    >>> b1.times_pressed
    2
    >>> b2.times_pressed
    3
    """

    def __init__(self, *args):
        _________________
        for _________ in _________________:
            _________________

    def press(self, info):
        """Takes in a position of the button pressed, and
        returns that button's output"""
        if ____________________:
            _________________
            _________________
            _________________
```

```
        ________________

    def typing(self, typing_input):
        """Takes in a list of positions of buttons pressed, and
        returns the total output"""
        ________________
        for ________ in ____________________:
            ________________
        ________________
```

Use Ok to test your code:

```
python3 ok -q Keyboard
```

Iterators and Generators

Q7: Pairs (generator)

Write a generator function `pairs` that takes a list and yields all the possible pairs of elements from that list.

```
def pairs(lst):
    """
    >>> type(pairs([3, 4, 5]))
    <class 'generator'>
    >>> for x, y in pairs([3, 4, 5]):
    ...     print(x, y)
    ...
    3 3
    3 4
    3 5
    4 3
    4 4
    4 5
    5 3
    5 4
    5 5
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q pairs
```

Q8: Pairs (iterator)

Now write an iterator that does the same thing. You are only allowed to use a linear amount of space - so computing a list of all of the possible pairs is not a valid answer. Notice how much harder it is - this is why generators are useful.

```
class PairsIterator:
    """
    >>> for x, y in PairsIterator([3, 4, 5]):
    ...     print(x, y)
    ...
    3 3
    3 4
    3 5
    4 3
    4 4
    4 5
    5 3
    5 4
    5 5
    """
    def __init__(self, lst):
        "*** YOUR CODE HERE ***"

    def __next__(self):
        "*** YOUR CODE HERE ***"

    def __iter__(self):
        "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q PairsIterator
```

Q9: Str

Write an iterator that takes a string as input and outputs the letters in order when iterated over.

```
class Str:
    """
    >>> s = Str("hello")
    >>> for char in s:
    ...     print(char)
    ...
    h
    e
    l
    l
    o
    >>> for char in s:    # a standard iterator does not restart
    ...     print(char)
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q Str
```

Linked Lists

Q10: Fold Left

Write the left fold function by filling in the blanks.

```
def foldl(link, fn, z):
    """ Left fold
    >>> lst = Link(3, Link(2, Link(1)))
    >>> foldl(lst, sub, 0) # (((0 - 3) - 2) - 1)
    -6
    >>> foldl(lst, add, 0) # (((0 + 3) + 2) + 1)
    6
    >>> foldl(lst, mul, 1) # (((1 * 3) * 2) * 1)
    6
    """
    if link is Link.empty:
        return z
    "*** YOUR CODE HERE ***"
    return foldl(______, ______, ______)
```

Use Ok to test your code:

```
python3 ok -q foldl
```

Q11: Fold Right

Now write the right fold function.

```
def foldr(link, fn, z):
    """ Right fold
    >>> lst = Link(3, Link(2, Link(1)))
    >>> foldr(lst, sub, 0) # (3 - (2 - (1 - 0)))
    2
    >>> foldr(lst, add, 0) # (3 + (2 + (1 + 0)))
    6
    >>> foldr(lst, mul, 1) # (3 * (2 * (1 * 1)))
    6
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q foldr
```

Q12: Filter With Fold

Write the `filterl` function, using either `foldl` or `foldr`.

```
def filterl(lst, pred):
    """ Filters LST based on PRED
    >>> lst = Link(4, Link(3, Link(2, Link(1))))
    >>> filterl(lst, lambda x: x % 2 == 0)
    Link(4, Link(2))
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q filterl
```

Q13: Reverse With Fold

Notice that `mapl` and `filterl` are not recursive anymore! We used the implementation of `foldl` and `foldr` to implement the actual recursion: we only need to provide the recursive step and the base case to `fold`.

Use `foldl` to write `reverse`, which takes in a recursive list and reverses it. *Hint*: It only takes one line!

Extra for experience: Write a version of `reverse` that do not use the `Link` constructor. You do not have to use `foldl` or `foldr`.

```
def reverse(lst):
    """ Reverses LST with foldl
    >>> reverse(Link(3, Link(2, Link(1))))
    Link(1, Link(2, Link(3)))
    >>> reverse(Link(1))
    Link(1)
    >>> reversed = reverse(Link.empty)
    >>> reversed is Link.empty
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q reverse
```

Q14: Fold With Fold

Write `foldl` using `foldr`! You only need to fill in the `step` function.

```
identity = lambda x: x

def foldl2(link, fn, z):
    """ Write foldl using foldr
    >>> list = Link(3, Link(2, Link(1)))
    >>> foldl2(list, sub, 0) # (((0 - 3) - 2) - 1)
    -6
    >>> foldl2(list, add, 0) # (((0 + 3) + 2) + 1)
    6
    >>> foldl2(list, mul, 1) # (((1 * 3) * 2) * 1)
    6
    """
    def step(x, g):
        "*** YOUR CODE HERE ***"
    return foldr(link, step, identity)(z)
```

Use Ok to test your code:

```
python3 ok -q foldl2
```