Lab 14: Final Review **lab14.zip (lab14.zip)**

Due by 11:59pm on Tuesday, December 1.

Starter Files

Download lab14.zip (lab14.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Required 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
```

Scheme

Q2: Split

Implement `split-at`, which takes a list `lst` and a positive 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`.

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

Use Ok to test your code:

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

Q3: 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

Q4: Num Splits

Given a list of numbers `s` and a target difference `d`, how many different ways are there to split `s` into two subsets such that the sum of the first is within `d` of the sum of the second? The number of elements in each subset can differ.

You may assume that the elements in `s` are distinct and that `d` is always non-negative.

Note that the order of the elements within each subset does not matter, nor does the order of the subsets themselves. For example, given the list `[1, 2, 3]`, you should not count `[1, 2], [3]` and `[3], [1, 2]` as distinct splits.

> Hint: If the number you return is too large, you may be double-counting somewhere. If the result you return is off by some constant factor, it will likely be easiest to simply divide/subtract away that factor.

```
def num_splits(s, d):
    """Return the number of ways in which s can be partitioned into two
    sublists that have sums within d of each other.

    >>> num_splits([1, 5, 4], 0)  # splits to [1, 4] and [5]
    1
    >>> num_splits([6, 1, 3], 1)  # no split possible
    0
    >>> num_splits([-2, 1, 3], 2) # [-2, 3], [1] and [-2, 1, 3], []
    2
    >>> num_splits([1, 4, 6, 8, 2, 9, 5], 3)
    12
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q num_splits
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Objects

Q5: Checking account

We'd like to be able to cash checks, so let's add a `deposit_check` method to our `CheckingAccount` class. It will take a `Check` object as an argument, and check to see if the `payable_to` attribute matches the `CheckingAccount` 's holder. If so, it marks the `Check` as deposited, and adds the amount specified to the `CheckingAccount` 's total.

Write an appropriate `Check` class, and add the `deposit_check` method to the `CheckingAccount` class. Make sure not to copy and paste code! Use inheritance whenever possible.

See the doctests for examples of how this code should work.

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals.

    >>> check = Check("Steven", 42)  # 42 dollars, payable to Steven
    >>> steven_account = CheckingAccount("Steven")
    >>> eric_account = CheckingAccount("Eric")
    >>> eric_account.deposit_check(check)  # trying to steal steven's money
    The police have been notified.
    >>> eric_account.balance
    0
    >>> check.deposited
    False
    >>> steven_account.balance
    0
    >>> steven_account.deposit_check(check)
    42
    >>> check.deposited
    True
    >>> steven_account.deposit_check(check)  # can't cash check twice
    The police have been notified.
    """
    withdraw_fee = 1
    interest = 0.01

    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)

    "*** YOUR CODE HERE ***"

class Check(object):
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q CheckingAccount
```

Tree Recursion

Q6: 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 between
    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
```

Linked Lists

Q7: 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
```

Q8: 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
```

Q9: 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
```

Q10: 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
```

CS 61A (/~cs61a/fa20/)

Weekly Schedule (/~cs61a/fa20/weekly.html)

Office Hours (/~cs61a/fa20/office-hours.html)

Staff (/~cs61a/fa20/staff.html)

Resources (/~cs61a/fa20/resources.html)

Studying Guide (/~cs61a/fa20/articles/studying.html)

Debugging Guide (/~cs61a/fa20/articles/debugging.html)

Composition Guide (/~cs61a/fa20/articles/composition.html)

Policies (/~cs61a/fa20/articles/about.html)

Assignments (/~cs61a/fa20/articles/about.html#assignments)

Exams (/~cs61a/fa20/articles/about.html#exams)

Grading (/~cs61a/fa20/articles/about.html#grading)