CS 61A
Spring 2023

Final Review
Discussion 13: April 26, 2023

Final Review

The following worksheet is final review! It covers various topics that have been seen throughout the semester.

Your TA will not be able to get to all of the problems on this worksheet so feel free to work through the remaining problems on your own. Bring any questions you have to office hours or post them on piazza.

Good luck on the final and congratulations on making it to the last discussion of CS61A!

Recursion

Q1: Paths List

(Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is [3, 4, 8, 9].

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    >>> paths(5, 3) # There is no valid path from x to y
    []
    """
    if x > y:
        return []
    elif x == y:
        return [[x]]
    else:
        a = paths(x + 1, y)
        b = paths(x * 2, y)
        return [[x] + subpath for subpath in a + b]
```

Mutation

Q2: Reverse

Write a function that reverses the given list. Be sure to mutate the original list. This is practice, so don't use the built-in `reverse` function!

```
def reverse(lst):
    """Reverses lst using mutation.

    >>> original_list = [5, -1, 29, 0]
    >>> reverse(original_list)
    >>> original_list
    [0, 29, -1, 5]
    >>> odd_list = [42, 72, -8]
    >>> reverse(odd_list)
    >>> odd_list
    [-8, 72, 42]
    """
    # iterative solution
    midpoint = len(lst) // 2
    last = len(lst) - 1
    for i in range(midpoint):
        lst[i], lst[last - i] = lst[last - i], lst[i]
```

Trees

Q3: Widest Level

Write a function that takes a `Tree` object and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]
    while x:
        levels.append([t.label for t in x])
        x = sum([t.branches for t in x], [])
    return max(levels, key=len)
```

```
I would definitely let students know about summing lists with the `sum` function and a
    default value along with reminding them about the key argument in the `max` function.
     Before throwing students into the function since it is conceptually hard, you could
    give them some conceptual starting point to spend less time on it.

**Main Idea** we'll traverse each level of the tree and keep track of the
elements of the levels. After we're done, we return the level with the most
items.

Here, `x` keeps track of the trees in the current level. To get the next
level of trees, we take all the branches from all the trees in the current
level. The special `sum` call is needed to make sure we get a list of
trees, instead of a list of branches (since branches are a list of trees
themselves).

Finally, we use `max` with a key to select the list with the longest
length from our list of levels.
```

Q4: In-order traversal

Write a function that returns a generator that generates an "in-order" traversal, in which we yield the value of every node in order from left to right, assuming that each node has either 0 or 2 branches.

```
def in_order_traversal(t):
    """
    Generator function that generates an "in-order" traversal, in which we
    yield the value of every node in order from left to right, assuming that each node
    has either 0 or 2 branches.

    For example, take the following tree t:
            1
        2       3
    4     5
         6  7

    We have the in-order-traversal 4, 2, 6, 5, 7, 1, 3

    >>> t = Tree(1, [Tree(2, [Tree(4), Tree(5, [Tree(6), Tree(7)])]), Tree(3)])
    >>> list(in_order_traversal(t))
    [4, 2, 6, 5, 7, 1, 3]
    """
    if t.is_leaf():
        yield t.label
    else:
        left, right = t.branches
        yield from in_order_traversal(left)
        yield t.label
        yield from in_order_traversal(right)
```

Linked Lists

Q5: Deep Map

Implement `deep_map`, which takes a function `f` and a `link`. It returns a *new* linked list with the same structure as `link`, but with `f` applied to any element within `link` or any `Link` instance contained in `link`.

The `deep_map` function should recursively apply `fn` to each of that `Link`'s elements rather than to that `Link` itself.

> *Hint*: You may find the built-in `isinstance` function for checking if something is an instance of an object. For example: >>> isinstance([1, 2, 3], list) True >>> isinstance(Link(1), Link) True >>> isinstance(Link(1, Link(2)), list) False

```
def deep_map(f, link):
    """Return a Link with the same structure as link but with fn mapped over
    its elements. If an element is an instance of a linked list, recursively
    apply f inside that linked list as well.

    >>> s = Link(1, Link(Link(2, Link(3)), Link(4)))
    >>> print(deep_map(lambda x: x * x, s))
    <1 <4 9> 16>
    >>> print(s) # unchanged
    <1 <2 3> 4>
    >>> print(deep_map(lambda x: 2 * x, Link(s, Link(Link(Link(5))))))
    <<2 <4 6> 8> <<10>>>
    """
    if link is Link.empty:
        return link
    if isinstance(link.first, Link):
        first = deep_map(f, link.first)
    else:
        first = f(link.first)
    return Link(first, deep_map(f, link.rest))
```

Generators

Q6: Repeated

Write a generator function that yields functions that are repeated applications of a one-argument function `f`. The first function yielded should apply `f` 0 times (the identity function), the second function yielded should apply `f` once, etc.

```
def repeated(f):
    """
    >>> double = lambda x: 2 * x
    >>> funcs = repeated(double)
    >>> identity = next(funcs)
    >>> double = next(funcs)
    >>> quad = next(funcs)
    >>> oct = next(funcs)
    >>> quad(1)
    4
    >>> oct(1)
    8
    >>> [g(1) for _, g in
    ...  zip(range(5), repeated(lambda x: 2 * x))]
    [1, 2, 4, 8, 16]
    """

    g = lambda x : x
    while True:
        yield g
        g = (lambda g: lambda x: f(g(x)))(g)
```

Scheme

Q7: Group by Non-Decreasing

Define a function `nondecreaselist`, which takes in a scheme list of numbers and outputs a list of lists, which overall has the same numbers in the same order, but grouped into lists that are non-decreasing.

For example, if the input is a stream containing elements

```
(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1)
```

the output should contain elements

```
((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1))
```

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

```
(define (nondecreaselist s)

    (if (null? s)
        nil
        (let ((rest (nondecreaselist (cdr s)) ))
            (if (or (null? (cdr s)) (> (car s) (car (cdr s))))
                (cons (list (car s)) rest)
                (cons (cons (car s) (car rest)) (cdr rest))
            )
        )
    )
)

(expect (nondecreaselist '(1 2 3 1 2 3)) ((1 2 3) (1 2 3)))

(expect (nondecreaselist '(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1))
        ((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1)))
```

SQL

The following questions will refer to two tables: - **records**: a table that stores information about the employees at a small company - **meetings**: a table which records the divisional meetings at the company

records

Name	Division	Title	Salary	Supervisor
Ben Bitdiddle	Computer	Wizard	60000	Oliver Warbucks
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
Cy D Fect	Computer	Programmer	35000	Ben Bitdiddle
Lem E Tweakit	Computer	Technician	25000	Ben Bitdiddle
Louis Reasoner	Computer	Programmer Trainee	30000	Alyssa P Hacker
Oliver Warbucks	Administration	Big Wheel	150000	Oliver Warbucks
Eben Scrooge	Accounting	Chief Accountant	75000	Oliver Warbucks
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge
...	...	...	...	...

meetings

Division	Day	Time
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm
...	...	...

Q8: Oliver Employee Meetings

Write a query that outputs the meeting days and times of all employees directly supervised by Oliver Warbucks.

```
SELECT m.day, m.time FROM records AS r, meetings AS m WHERE r.division = m.division
AND r.supervisor = "Oliver Warbucks";
```

Q9: Different Division

Write a query that outputs the names of employees whose supervisor is in a different division.

```
SELECT e.name FROM records AS e, records AS s WHERE e.supervisor = s.name AND e.division
    != s.division;
```

Q10: Num Meetings

Write a query that outputs the days of the week for which fewer than 5 employees have a meeting. You may assume no department has more than one meeting on a given day.

```
SELECT m.day FROM records AS e, meetings AS m WHERE e.division = m.division GROUP BY m.
    day HAVING COUNT(*) < 5;
```