CS 61A
Fall 2021

Final Review
Discussion 14: December 1, 2021

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]]
    """
    if ____________________________________
        return ____________________________________
    elif ____________________________________
        return ____________________________________
    else:
        a = ____________________________________
        b = ____________________________________
        return ____________________________________
```

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]
    """
    "*** YOUR CODE HERE ***"









# You can use more space on the back if you want
```

Trees

Q3: Reverse Other

Write a function `reverse_other` that mutates the tree such that **labels** on *every other* (odd-depth) level are reversed. For example, `Tree(1,[Tree(2, [Tree(4)]), Tree(3)])` becomes `Tree(1,[Tree(3, [Tree(4)]), Tree(2)])`. Notice that the nodes themselves are *not* reversed; only the labels are.

```
def reverse_other(t):
    """Mutates the tree such that nodes on every other (odd-depth)
    level have the labels of their branches all reversed.

    >>> t = Tree(1, [Tree(2), Tree(3), Tree(4)])
    >>> reverse_other(t)
    >>> t
    Tree(1, [Tree(4), Tree(3), Tree(2)])
    >>> t = Tree(1, [Tree(2, [Tree(3, [Tree(4), Tree(5)]), Tree(6, [
    Tree(7)])]), Tree(8)])
    >>> reverse_other(t)
    >>> t
    Tree(1, [Tree(8, [Tree(3, [Tree(5), Tree(4)]), Tree(6, [Tree(7)
    ])]), Tree(2)])
    """
    "*** YOUR CODE HERE ***"

# You can use more space on the back if you want
```

Linked Lists

Q4: 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.

```
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>>>
    """
    "*** YOUR CODE HERE ***"









# You can use more space on the back if you want
```

Generators

Q5: 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 = ______________________________________________
    while True:
        ______________________________________________
        ______________________________________________
```

Scheme

Q6: 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 _____________________________
        _____________________________
        (let ((rest _____________________________ ))
            (if _____________________________
                (cons _____________________________ rest)
                (cons _____________________________ (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)))
```

Regex

Q7: Greetings

Let's say hello to our fellow bears! We've received messages from our new friends at Berkeley, and we want to determine whether or not these messages are *greetings*. In this problem, there are two types of greetings - salutations and valedictions. The first are messages that start with "hi", "hello", or "hey", where the first letter of these words can be either capitalized or lowercase. The second are messages that end with the word "bye" (capitalized or lowercase), followed by either an exclamation point, a period, or no punctuation. Write a regular expression that determines whether a given message is a greeting.

```
import re

def greetings(message):
    """
    Returns whether a string is a greeting. Greetings begin with
    either Hi, Hello, or
    Hey (either capitalized or lowercase), and/or end with Bye (
    either capitalized or lowercase) optionally followed by
    an exclamation point or period.

    >>> greetings("Hi! Let's talk about our favorite submissions to
    the Scheme Art Contest")
    True
    >>> greetings("Hey I just figured out that when I type the
    Konami Code into cs61a.org, something fun happens")
    True
    >>> greetings("I'm going to watch the sun set from the top of
    the Campanile! Bye!")
    True
    >>> greetings("Bye Bye Birdie is one of my favorite musicals.")
    False
    >>> greetings("High in the hills of Berkeley lived a legendary
    creature. His name was Oski")
    False
    >>> greetings('Hi!')
    True
    >>> greetings("bye")
    True
    """
    return bool(re.search(__________, message))
```

BNF

Q8: Comprehension is Everything

(Adapted from Spring 2021 Final) The following EBNF grammar can describe a subset of Python list comprehensions, but cannot yet describe all of them.

```
start: comp

?comp: "[" expression "for" IDENTIFIER "in" IDENTIFIER "]"

expression: IDENTIFIER operation*

operation: OPERATOR NUMBER

IDENTIFIER: /[a-zA-Z]+/

OPERATOR: "*" | "/" | "+" | "-"

%import common.NUMBER
%ignore /\s+/
```

Select all of the non-terminal symbols in the grammar:

- `comp`
- `expression`
- `operation`
- `NUMBER`
- `IDENTIFIER`
- `OPERATOR`

Which of the following comprehensions would be successfully parsed by the grammar?

- `[ x * 2 for x in list ]`
- `[ x for x in list ]`
- `[ x ** 2 for x in list ]`
- `[ x + 2 for x in list if x == 1 ]`
- `[ x * y for x in list for y in list2 ]`
- `[ x - 2 for x in my_list ]`
- `[ x - y for (x,y) in tuples ]`

Which line would we need to modify to add support for a `%` operator, like in the expression `[ n % 2 for n in numbers ]`?

- `OPERATOR: "*" | "/" | "+" | "-"`
- `IDENTIFIER: /[a-zA-z]+/`
- `operation: OPERATOR NUMBER`
- `expression: IDENTIFIER operation*`
- `?comp: "[" expression "for" IDENTIFIER "in" IDENTIFIER "]"`

SQL

(Adapted from Fall 2019) The scoring table has three columns, a player column of strings, a points column of integers, and a quarter column of integers. The players table has two columns, a name column of strings and a team column of strings. Complete the SQL statements below so that they would compute the correct result even if the rows in these tables were different than those shown.

Important: You may write anything in the blanks including keywords such as WHERE or ORDER BY. Use the following tables for the questions below:

```
CREATE TABLE scoring AS
    SELECT "Donald Stewart" AS player, 7 AS points, 1 AS quarter
    UNION
    SELECT "Christopher Brown Jr.", 7, 1 UNION
    SELECT "Ryan Sanborn", 3, 2 UNION
    SELECT "Greg Thomas", 3, 2 UNION
    SELECT "Cameron Scarlett", 7, 3 UNION
    SELECT "Nikko Remigio", 7, 4 UNION
    SELECT "Ryan Sanborn", 3, 4 UNION
    SELECT "Chase Garbers", 7, 4;

CREATE TABLE players AS
    SELECT "Ryan Sanborn" AS name, "Stanford" AS team UNION
    SELECT "Donald Stewart", "Stanford" UNION
    SELECT "Cameron Scarlett", "Stanford" UNION
    SELECT "Christopher Brown Jr.", "Cal" UNION
    SELECT "Greg Thomas", "Cal" UNION
    SELECT "Nikko Remigio", "Cal" UNION
    SELECT "Chase Garbers", "Cal";
```

Q9: Big Quarters

Write a SQL statement to select a one-column table of quarters in which more than 10 total points were scored.

Q10: Score

Write a SQL statement to select a two-column table where the first column is the team name and the second column is the total points scored by that team. Assume that no two players have the same name.