CS 61A Fall 2020

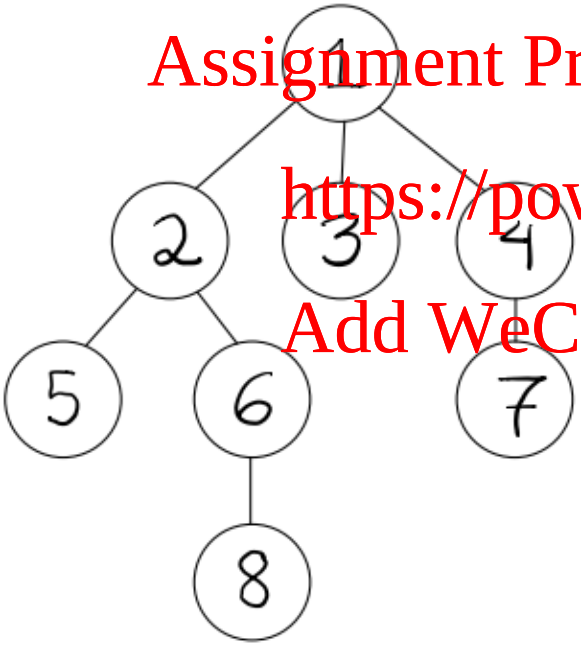
Structure and Interpretation of Computer Programs

Quiz 7

INSTRUCTIONS

- Please review this worksheet before the exam prep session. Coming prepared will help greatly, as the TA will be live solving without allocating much time for individual work.
- Either Sean or Derek will be on video live solving these questions. The other TA will be answering questions in the chat. It is in your best interest to come prepared with **specific** questions.
- This is not graded, and you do not need to turn this in to anyone.

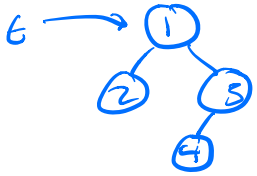
Below is a tree, which will be referred to as `t1` in future questions.



1. **Node Function Generator**

(a) Construct the generator function `node_func_gen`, which takes in a tree `t`, a one-argument filter function `f` and a one-argument mapping function `g`. `node_func_gen` yields one *positional node function* for each node position in the `t`. A positional node function takes in a tree as an argument and returns the value of the node at the specified position. For example, a positional node function constructed for the root of a tree will return the label of the root of any tree that it is given. You may assume that any trees passed to your positional functions have the same structure as `t`.

You may not use `list` or any sort of comprehension.



$f_4(t) \rightarrow 4$

```
def node_func_gen(t, f, g):
    """
    >>> f = lambda x: x % 2 == 1
    >>> g = lambda x: x**2
    >>> for func in node_func_gen(t1, f, g) # note: order doesn't matter
    ...     print(func(t1))
    ...
    1
    25
    9
    49
    """
    if _____:
        yield _____
    for i in range(len(t.branches)):
        yield from _____
```

lambda f: lambda t: f(t.branches[i])

f(t.label)

lambda x: g(x.label)

map((lambda i: lambda f: lambda t: f(t.branches[i]))(i), node_func_gen(t.branches[i], f, g))

(function, iterable)

(lambda i: lambda f: lambda t: f(t.branches[i]))(i)

Prep

Base case

When do you yield a value? Say f only keeps even values.

Once we decide to yield a value, how should we modify it?

↳ use g!

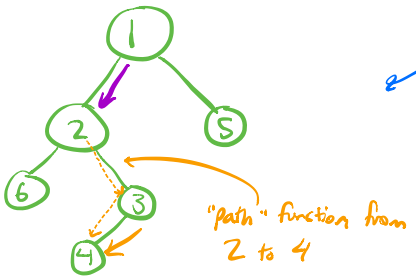
```
if f(t.label):
    yield g(t.label)
```

How about in the recursive case?

↳ What do we get when we recurse on a branch? (Leap of Faith)

↓ we get a generator of functions which take us from that child node to our filtered, mapped node values.

↓ All we have to do is modify these functions so they take you from the root to the child, then recursion will take you the rest of the way.



So we want to map this modification onto all recursively yielded functions

recursion:

```
map(?, node_func_gen(t.branches[i], f, g))
```

should take in a function f and return a positional node function from root of t to target

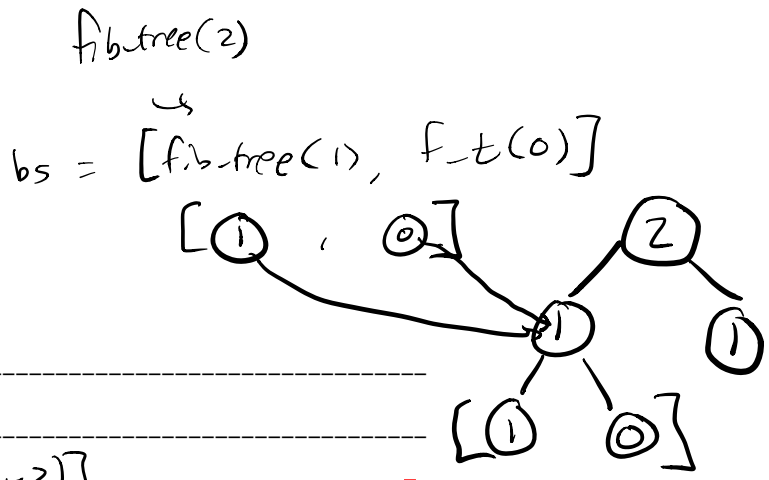
needs to map t to i^{th} branch first

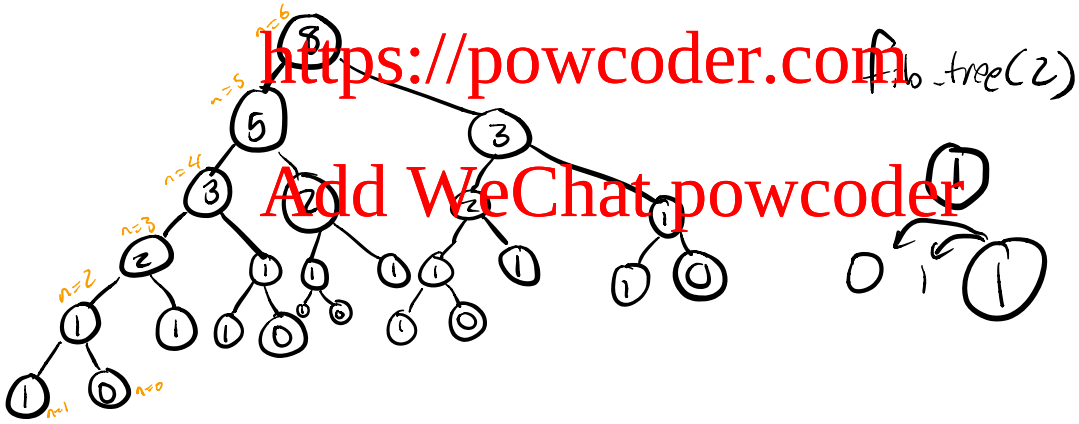
```
(lambda i: lambda f: lambda t: f(t.branches[i]))(i)
```

2. **Fibonacci Tree**

Implement `fib_tree`, which takes in a non-negative integer `n` and returns a depth-`n` *Fibonacci tree* of depth n. The root label of a depth-`n` Fibonacci tree is the `n`th Fibonnaci number (indexed from zero), and each node in the tree has exactly two children which contain the values needed to compute the Fibonacci number of their parent. If no lower Fibonacci numbers are required to compute the value of a node, then the node is a leaf.

```
def fib_tree(n):
    """
    >>> t = fib_tree(6)
    >>> t.label
    8
    >>> t.branches[0].label
    5
    >>> t.branches[1].label
    3
    """
    if _____
        return _____
    bs = _____
    return _____
```



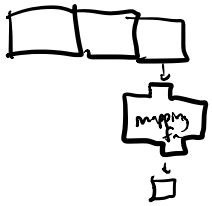


yield from x
yield from y
⟺
for item in x:
    yield item
for item in y:
    yield item

3. **Subsequence Generator**

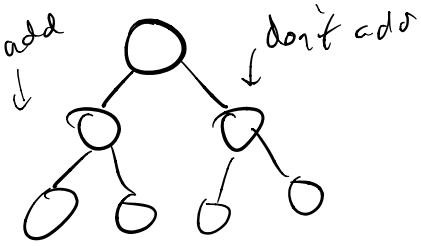
(a) Implement `subsequences`, which takes in a list `lst` and yields all subsequences of `lst`. You may not use the `list` function or any comprehensions.

```
def subsequences(lst):
    """
    >>> for seq in subsequences([1, 2, 3]) # note: order doesn't matter
    ...     print(seq)
    ...
    [1, 2, 3]
    [1, 2]
    [1, 3]
    [1]
    [2, 3]
    [2]
    [3]
    []
    """
    if _____:
        case 1: yield from _____
        case 2: yield from _____
    else:
        yield _____
```



case 1: include first item
case 2: don't include first item
↳ don't need to modify recursive results
↳ modify results to include 1 at beginning

lst
lst[0]
lst[1:]
puts lst[0] onto the front of all results
discards lst[0]
[]
we know lst = []



subsequences([1, 2])
[1, 2]
[1]
[2]
[]
subsequences([2])
yield []
subsequences([])
[2, 3]
[2]
[3]
[]

4. **Minimax**

You are playing a game with a friend. At the end of the game you are both given the same score. Your friend wants to make this score as low as possible, and you want to make this score as high as possible. This situation can be modeled with a tree: leaf nodes correspond to the game being over and contain the score at the end of the round. The values of intermediate nodes don't matter.

Both you and your friend have perfect knowledge of this game tree `t`. With this setup, implement `first_move`, which takes in a game tree `t` as input and outputs the *index* of the branch in the tree corresponding to your optimal first move.

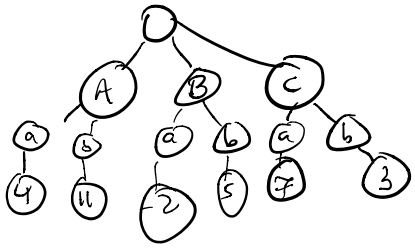
Hint: all even-depth layers correspond to your turns, and all odd-depth layers correspond to your friend's turns.

```
def first_move(t):
    """
    >>> first_move(t1)
    2
    >>> # best choice is the rightmost child (value is 7)
    >>> # index of rightmost child in t.branches is 2
    """

    def optim_val(tr, fn=min):
        if tr.is_leaf():
            return tr.label
        return fn([optim_val(b, min if fn is max else max) for b in tr.branches])

    return max(range(len(t.branches)), key=lambda i: optim_val(t.branches[i]))
```

max or min (probably)




```
def max_score(t):
    return t.label if t.is_leaf()
        else max([max_score(b) for
                  b in t.branches])
```