CS 61A Summer 2020

Nonlocal, Midterm Review

Discussion 6: July 14, 2020

1 Nonlocal

Until now, you've been able to access names in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a binding in a parent frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal name
        num = num + 1 # modifies num in the stepper frame
        return num
    return step

>>> step1 = stepper(10)
>>> step1()                # Modifies and returns num
11
>>> step1()                # num is maintained across separate calls to step
12
>>> step2 = stepper(10)    # Each returned step function keeps its own state
>>> step2()
11
```

As illustrated in this example, `nonlocal` is useful for maintaining state across different calls to the same function.

However, there are two important caveats with `nonlocal` names:

- **Global names** cannot be modified using the `nonlocal` keyword.
- **Names in the current frame** cannot be overridden using the `nonlocal` keyword. This means we cannot have both a local and nonlocal binding with the same name in a single frame.

Because `nonlocal` lets you modify bindings in parent frames, we call functions that use it **mutable functions**.

Questions

1.1 Draw the environment diagram for the following code.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

1.2 Write a function that takes in a number `n` and returns a one-argument function. The returned function takes in a function that is used to update `n`. It should return the updated `n`.

```
def memory(n):
    """
    >>> f = memory(10)
    >>> f(lambda x: x * 2)
    20
    >>> f(lambda x: x - 7)
    13
    >>> f(lambda x: x > 5)
    True
    """
```

2 Midterm Review

For any problems that may require it, the tree ADT is provided at the end of this packet for your convenience. This section is **far** longer than a typical discussion, and it is recommended that you also use it as a problem bank for your midterm studies! Best of luck, you got this!!

Questions

2.1 Draw the environment diagram that results from executing the code below.

```
from operator import add

six = 1

def ty(one, a):
    summer = one(a, six)
    return summer

six = ty(add, 6)
summer = ty(add, 6)
```

2.2 Write a function that takes in no arguments and returns two functions, `prepend` and `get`, which represent the "add to front of list" and "get the ith item" operations, respectively. Do not use any python built-in data structures like lists or dictionaries. You do not necessarily need to use all the lines.

```
def nonlocalist():
    """
    >>> prepend, get = nonlocalist()
    >>> prepend(2)
    >>> prepend(3)
    >>> prepend(4)
    >>> get(0)
    4
    >>> get(1)
    3
    >>> get(2)
    2
    >>> prepend(8)
    >>> get(2)
    3
    """
    get = lambda x: "Index out of range!"
    def prepend(value):
        ____________________________________________________
        f = ____________________________________________________
        def get(i):
            if i == 0:
                return value
            return ____________________(_________________________)
        ____________________________________________________
    return _____________________________, _____________________________
```

2.3 Fill in the definition of `f` below such that the interpreter prints as expected. **Your solution must be on one line**.

```
>>> f = ______________________________________________
>>> f = f(10)
1
2
3
4
5
6
7
8
9
10
```

Then, given your definition of `f`, what will be printed below? (Assuming that the above lines have also been executed in the interpreter.)

```
>>> f
```

2.4 (Spring 2015) Implement the `memory` function, which takes a number `x` and a single-argument function `f`. It returns a function with a peculiar behavior that you must discover from the doctests. You may only use names and call expressions in your solution. You may not write numbers or use features of Python not yet covered in the course.

```
square = lambda x: x * x
double = lambda x: 2 * x
def memory(x, f):
    """Return a higher-order function that prints its
    memories.
    >>> f = memory(3, lambda x: x)
    >>> f = f(square)
    3
    >>> f = f(double)
    9
    >>> f = f(print)
    6
    >>> f = f(square)
    3
    None
    """
    def g(h):

        print(_____________________________________)

        return _____________________________________

    return g
```

2.5 It's Hog again! Write a commentary function `announce_losses` that takes in a player `who` and returns a commentary function that announces whenever that player loses points.

```
def announce_losses(who, last_score=0):
    """
    >>> f = announce_losses(0)
    >>> f1 = f(10, 0)
    >>> f2 = f1(1, 10) # Player 0 loses points due to swine swap
    Oh no! Player 0 just lost 9 point(s).
    >>> f3 = f2(7, 10)
    >>> f4 = f3(7, 11) # Should not announce when player 0's score does not change
    >>> f5 = f4(11, 12)
    """
    assert who == 0 or who == 1, 'The who argument should indicate a player.'
    def say(score0, score1):
        if who == 0:
            score = ________________________
        elif who == 1:
            score = ________________________
        if ________________________:
            ____________________________________________________
        return ________________________
    return say
```

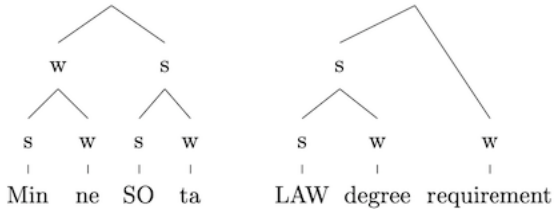
2.6 (Fall 2013) The CS 61A staff has developed a formula for determining what a fox might say. Given three strings—a start, a middle, and an end—a fox will say the start string, followed by the middle string repeated a number of times, followed by the end string. These parts are all separated by single hyphens.

Complete the definition of `fox_says`, which takes the three string parts of the fox's statement (start, middle, and end) and a positive integer `num` indicating how many times to repeat middle. It returns a string. You cannot use any for or while statements. Use recursion in repeat. Moreover, you cannot use string operations other than the + operator to concatenate strings together.

```
def fox_says(start, middle, end, num):
    """
    >>> fox_says('wa', 'pa', 'pow', 3)
    'wa-pa-pa-pa-pow'
    >>> fox_says('fraka', 'kaka', 'kow', 4)
    'fraka-kaka-kaka-kaka-kaka-kow'
    """
    def repeat(k):
```

```
    return start + '-' + repeat(num) + '-' + end
```

2.7 The study of stress is still an open field of inquiry in linguistics—why do we say "alaBAma," but "aLAbama" and "alabaMA" make us cringe? Or how did it come to be that "AMERICAN history professor" and "american HISTORY professor" mean two different things? One model that we use to understand stress actually employs the tree data structure!



In the above diagrams, every node has a "strong" child and a "weak" child, and primary stress is placed on the leaf that has the greatest number of strong parents. In the spirit of computational linguistics, let's write a function that, given one of these tree structures, identifies the stressed part of a word or phrase.[1]

```
def primary_stress(t):
    """
    >>> word = tree("", [
        tree("w", [tree("s", [tree("min")]), tree("w", [tree("ne")])]),
        tree("s", [tree("s", [tree("so")]), tree("w", [tree("ta")])])])
    >>> primary_stress(word)
    'so'
    >>> phrase = tree("", [
        tree("s", [tree("s", [tree("law")]), tree("w", [tree("degree")])]),
        tree("w", [tree("requirement")])])
    >>> primary_stress(phrase)
    'law'
    """
    def helper(t, num_s):
        if is_leaf(t):
            return [label(t), num_s]
        if label(t) == "s":
            num_s = _________________________
        return max([_____________________________________________],
            key = _________________________)
    return _________________________
```

[1] Inspiration for this problem comes from *Liberman, Mark and Alan Prince. 1977. On stress and linguistic rhythm. Linguistic Inquiry. 8:249-336.*, and from the course Linguistics 111 (Phonology).

2.8 Consider the subset sum problem: you are given a list of integers and a number k. Is there a subset of the list that adds up to k? For example:

```
>>> subset_sum([2, 4, 7, 3], 5)         # 2 + 3 = 5
True
>>> subset_sum([1, 9, 5, 7, 3], 2)
False
>>> subset_sum([1, 1, 5, -1], 3)
False
```

Note: You can use the `in` operator to determine if an element belongs to a list:

```
>>> 3 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
False
```

```
def subset_sum(seq, k):
```

3 Tree ADT

```
# Constructor
def tree(label, branches=[]):
    """Construct a tree with the given label value and a list of branches."""
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

# Selector
def label(tree):
    """Return the label value of a tree."""
    return tree[0]

# Selector
def branches(tree):
    """Return the list of branches of the given tree."""
    return tree[1:]

def is_tree(tree):
    """Returns True if the given tree is a tree, and False otherwise."""
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

# For convenience
def is_leaf(tree):
    """Returns True if the given tree's list of branches is empty, and False
    otherwise.
    return not branches(tree)
```