CS 61A Mutable Trees and Mutable Functions
Summer 2018

Discussion 9: July 19, 2018

1 Mutable Trees

Recall the tree abstract data type: a tree is defined as having a label and some branches. Previously, we implemented the tree abstraction using Python lists. Let's look at another implementation using objects instead:

```
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

Notice that with this implementation we can mutate a tree using attribute assignment, which wasn't possible in the previous implementation using lists.

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```

Questions

1.1 What would Python display? If you believe an expression evaluates to a *Tree* object, write *Tree*.

```
>>> t0 = Tree(0)
>>> t0.label


>>> t0.branches


>>> t1 = Tree(0, [1, 2]) # Is this a valid tree?


>>> t2 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
>>> t2.branches[0]


>>> t2.branches[1].branches[0].label
```

1.2 Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
```

1.3 Define a function `square_tree(t)` that squares every value in the non-empty tree `t`. You can assume that every value is a number.

```
def square_tree(t):
    """Mutates a Tree t by squaring all its elements."""
```

1.4 Assuming that every value in `t` is a number, let's define `average(t)`, which returns the average of all the values in `t`.

```
def average(t):
    """
    Returns the average value of all the nodes in t.
    >>> t0 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
    >>> average(t0)
    1.5
    >>> t1 = Tree(8, [t0, Tree(4)])
    >>> average(t1)
    3.0
    """
```

2 Mutable Functions in Python

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

2.1 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()
```

2.2 Given the definition of `make_shopkeeper` below, draw the environment diagram.

```
def make_shopkeeper(total_gold):
    def buy(cost):
        nonlocal total_gold
        if total_gold < cost:
            return 'Go farm some more champions'
        total_gold = total_gold - cost
        return total_gold
    return buy

infinity_edge, zeal, gold = 3800, 1100, 3800
shopkeeper = make_shopkeeper(gold - 1000)
shopkeeper(zeal)
shopkeeper(infinity_edge)
```

2.3 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 prints the updated `n` value and returns a function that has the same behavior as itself.

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

2.4 The bathtub below simulates an epic battle between Finn and Kylo Ren over a populace of rubber duckies. Fill in the body of `ducky` so that all doctests pass.

```
def bathtub(n):
    """
    >>> annihilator = bathtub(500) # the force awakens...
    >>> kylo_ren = annihilator(10)
    >>> kylo_ren()
    490 rubber duckies left
    >>> rey = annihilator(-20)
    >>> rey()
    510 rubber duckies left
    >>> kylo_ren()
    500 rubber duckies left
    """
    def ducky_annihilator(rate):
        def ducky():









        return ducky
    return ducky_annihilator
```

3 Mutable Functions in Scheme

We can also create mutable functions in Scheme. In Python, when we want to modify a binding in a parent frame, we declare it to be `nonlocal` at the start of the function and then assign to the name as normal.

In Scheme, we don't need to declare which bindings from a parent frame we wish to modify. Instead, we use a new special form call `set!` when we want to modify an existing binding (regardless of whether that binding exists in the current frame or a parent frame).

Just like the `define` special form, `set!` takes in two operands: the symbol to be re-assigned and an expression that should be evaluated and assigned to that symbol.

```
(set! <symbol> <expression>)
```

Here's the same `stepper` function from earlier, now written in Scheme.

```
(define (stepper num)
  (define (step)
    (set! num (+ num 1))
    num)
  step)
```

`set!` will always modify the most local binding for that symbol that exists. In other words, if the symbol is bound in the current frame, `set!` works the same as `define`. Otherwise, it proceeds through parent frames until it finds the symbol, and then re-binds it in that frame. If the binding does not exist anywhere within the current environment, `set!` will error.

Unlike `nonlocal`, `set!` can even modify bindings in the global frame. For example:

```
scm> (define count 0)
count
scm> (define (increment) (set! count (+ count 1)))
increment
scm> (increment)
scm> (increment)
scm> (increment)
scm> count
3
```

Questions

3.1 Write a procedure `make-piggy-bank`, which returns a one-argument procedure (which we'll call a piggy bank). A piggy bank starts with nothing inside it. Each time you call it with a positive number, that amount is added to the bank's total. If you instead pass in number less than or equal to 0, the piggy bank should reset its total to 0 and then return the old total.

```
(define (make-piggy-bank)
```

```
scm> (define piggy-bank (make-piggy-bank))
piggy-bank
scm> (piggy-bank 5)  ; add $5
scm> (piggy-bank 10) ; add $10
scm> (piggy-bank 3)  ; add $3
scm> (piggy-bank 0)  ; dump the money out
18
scm> (piggy-bank 4)  ; add $4
scm> (piggy-bank 0)  ; dump the money out
4
```