CS61A Lecture 38

Wednesday, December 4, 2019

Announcements

- There will be video lectures on Friday as usual.
- Guerilla section on streams and SQL on Saturday 12/7 at 12-2pm.
- CSM will be running review sessions during regular lecture time.

Final Review

"There is a very low chance the [exact] problems we discuss today will be on the final exam."
– Professor DeNero, 2019

Tree-Structured Data

We learned two ways of representing tree-structured data: ADT and OOP. The tree functionality is not built into Python by default, so as to be modifiable as needed for the purpose it is needed.

```
def tree(label, branches=[]):
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_leaf(t):
    return not branches(t)

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
    def is_leaf(self):
        return not self.branches
```

Trees can contain other trees! A prominent example of a tree is HTML, which has a hierarchial structure like a tree.

Tree Processing

There is a very simple checklist we can use to solve data in a tree structure, and we will use an example to demonstrate:

> **Example**
> Implement `bigs`, which takes a Tree instance t containing integer labels. It returns the number of nodes in t whose labels are larger than all labels of their ancestor nodes.
>
> ```
> ```

```
def bigs(t):
    """Return the number of nodes in T that are larger than all their ancestors.
    >> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]) Tree(3, [Tree(0, [Tree(2)])])])
    """
    ...
```

Well, first, we should ignore the starter code and figure out what the question is asking for. Draw the examples, especially for a tree problem, as visualizing will help you figure out what works and what doesn't.

```
        1
      /   \
    4       3
  /   \      |
4     5      1
             |
             0
```

Our usual structure for solving tree problems is as such:

```
    if t.is_leaf():
        return ___
    else:
        return ___([___ for b in t.branches])
```

It doesn't quite work here, so try and figure out if there's a better solution for yourself.

There are two possible general approaches, which are:

```
    if node.label > max(ancestors):
        #here you track the list of ancestors
    if node.label > max_ancestor:
        #here you track only the biggest ancestor
```

So here's the skeleton code this question would've been accompanied by:

```
def bigs(t):
    def f(a,x):
        if ___________:
            return 1 + ___________
        else:
            return ___________
    return ___________
```

We know the base case from our previous analysis, and we also know we need to call the helper function:

```
def bigs(t):
    def f(a,x):
        if a.label > x:
            return 1 + ___________
        else:
```

```
            return ____________
    return f(t,...)
```

Now, we can start designing the recursive call:

```
def bigs(t):
    def f(a,x):
        if a.label > x:
            return 1 + sum([f(b,a.label) for b in a.branches])
        else:
            return sum([f(b,x) for b in a.branches])
    return f(t,...)
```

And our final step is that we need to answer the final blank above! What's the initial value for the largest ancestor so far?

```
def bigs(t):
    def f(a,x):
        if a.label > x:
            return 1 + sum([f(b,a.label) for b in a.branches])
        else:
            return sum([f(b,x) for b in a.branches])
    return f(t,t.label-1)
```

Now, you should double-check your solutions to ensure you got the right solution, and make sure you're passing the right things into your recursive calls too.

Recursive Accumulation

It is often the case you need to think of more than one way to solve the same problem. Here's a different implementation of `bigs`:

```
def bigs(t):
    n = 0
    def f(a,x):
        nonlocal n
        if a.label > x:
            n+=1
        for b in a.branches:
            f(b,max(a.label,x))
    f(t,t.label-1)
    return n
```

How to Design Programs

According to the MIT textbook, "How to Design Programs", here are the exact steps you should take to design a program:

From Problem Analysis to Data Definitions

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header

State what kind of data the desired function consumes and produces. Formulate a concise answer to the question what the function computes. Define a stub that lives up to the signature.

Functional Examples

Work through examples that illustrate the function's purpose.

Function Template

Translate the data definitions into an outline of the function.

Function Definition

Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Testing

Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Implementing the Design Process

Implement `smalls`, which takes a Tree instance t containing integer labels. It returns the non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.

Watch the video to see it in action.

Interpreter Analysis

The final review will involve revisiting the Scheme project.

How many times does scheme_eval get called when evaluating the following expressions?

```
(define x (+ 1 2))

(define (f y) (+ x y))

(f (if (> 3 2) 4 5))
```

Solution

Interpreter Analysis

How many times does scheme_eval get called when evaluating the following expressions?