CSC148 Lab#6, summer 2016

learning goals

In this lab you will practice tracing and implementing recursive functions where the input is a non-list, a possibly-nested list, or a general tree as presented in lecture.

Note that this is a long lab and you should work on these on your own actually before going to the lab. You are encouraged to then go to the lab where you can get guidance and feedback from your TA and other classmates. There will be a short quiz during the last 15 minutes of the lab.

Part 1. Writing recursion (on possibly-nested list)

Open file nested_list.py and save it under a new sub-directory called `lab06`. This file provides you headers and docstrings for the functions you will implement, as well as a helper function we think you'll find useful:

`gather_lists`: There will be cases where you have a list whose elements are sub-lists, and what you'd really like is to concatenate them into a single list. That's what `gather_lists` does.

Now implement the following functions.

implement `list_all`

Read over the header and docstring for this function, but don't write any implementation until you fill in the steps below:

1. One of the examples in our docstring is simple enough not to require recursion. Write out an `if` condition that checks for such cases, and then returns the correct thing. Include an `else`... for when the argument is not so easy to deal with.
2. Now suppose the function works correctly and does what the docstring claims: returns a list of the non-list values from the list or non-list it is given. If you call the function on each element in a list and it returns such a list, how will you combine the resulting list of lists into the correct output?

Go over these parts with your TA. After that, fill in the implementation and see whether it works.

implement `max_length`

Read over the docstring but again, don't write the implementation before working through the steps below.

1. In the docstring there is an example where the argument can't be broken into smaller, similar parts. Write an `if` condition that detects such cases, and then returns the right thing. Add an `else` to deal with lists that can be broken into parts that can be dealt with recursively.
2. Now assume the function works properly when it is called on the elements of a list: for a list, it returns the maximum length of the list and all its sublists, or for a non-list it returns 0. If you call the function and get a correct result for each element of a list, how can you correctly combine them so the function gives the right result for the entire list? (Don't forget the length of the entire list figures into your result).

Go over these parts with your TA. After that, fill in the implementation, and see whether it works.

implement `list_over`

Read over the docstrings, then complete the steps below:

1. There are some arguments that cannot be decomposed into parts that can be solved by this same function. Write `if` and `elif` conditions to detect these cases, and then return the correct values. Put an `else` in place for the remaining cases, where the argument can be decomposed into recursive subcases.
2. Assume that the function now satisfies the docstring for all the elements of a list: if the element is string of length over n, it produces a list containing that number; it the element is another string, it produces an empty list, and if it is a list of strings, it produces a list of those over length n. If the function produced the correct result for each element of a list, how would you combine all those results into the correct overall result?

Go over these parts with your TA, then implement the function and try it out on the given examples and any others you think of.

Part 2. Working with general trees

set-up

Open file tree.py and save it under sub-directory lab06. This file provides you with a declaration of class Tree, headers and docstrings for the functions you will implement, as well as two utility functions we think you'll find useful:

`gather_lists`: There will be cases where you have a list whose elements are sub-lists, and what you'd really like is to concatenate them into a single list. That's what gather lists does. You'll need to save csc148_queue.py in lab06 to make this work.

`Descendants_from_list`: This function makes it less laborious to build a Tree from a list of data. It simply fills in the tree, level by level, from a list, until it runs out of elements.

Once you have familiarized yourself with the `__init__` method for class Tree (as well as other methods for the same class), you are ready to proceed to the implementation of the functions below. If you have questions, call your TA over.

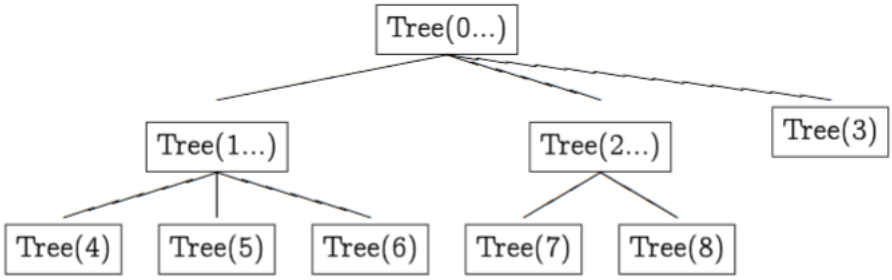
implement `list_internal`

This function will list all the values from internal nodes of Tree t. The order of the list is not specified, and there is no need to remove duplicates.

Read over the header and docstring for this function in `tree.py`, but don't write any implementation until you fill in the steps below:

1. One of the examples in our docstring is simple enough not to require recursion. Write out an `if`... expression that checks for this case, and then returns the correct thing. Include an `else`... for when the tree is not so easy to deal with.
2. Below is a picture of a larger Tree, with several levels. Consider a function call `list_internal(t)`, assuming that t is a reference to the Tree. Are there one or more smaller

trees for which it would be helpful to know the list of values that they contain? Which smaller trees are they? Write an example of a function call list internal on one of these smaller trees. You can access these trees through the variable t.



3. Suppose the call in the previous step gives you the correct answer according to the docstring: it returns a list of the values in the tree it is given. How will you combine the solutions for all the smaller instances to get a solution for Tree t itself? Write code to return the correct thing. Hint: You may want `gather_lists` here. Put this code in the else... expression that you created in the first step.

Go over these three parts with your TA. After that, fill in the implementation in tree.py, and see whether it works. A good approach is to comment-out all the functions you have not yet implemented, so that you will see only doctest results for the one you're working on.

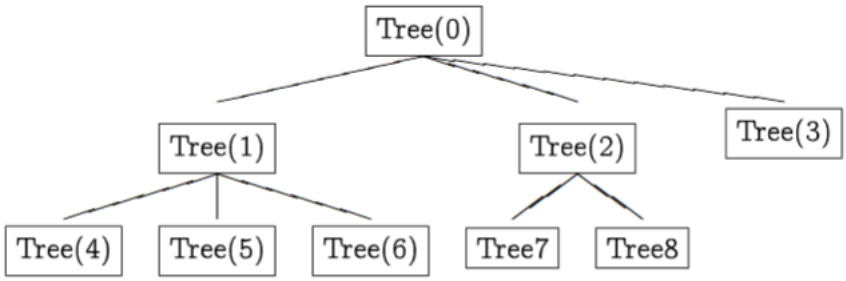
arity

This function returns the maximum branching factor of this tree.

Read over the header and docstring for this function in `tree.py`, but don't write any implementation until you fill in the steps below:

1. One of the examples in our docstring is simple enough not to require recursion. Write out an `if`... expression that checks for this case, and then returns the correct thing. Include an `else`... for when the tree is less easy to deal with.

2. Below is a picture of a larger Tree t, with several levels. Consider a function call that passes t as an argument. Are there one or more smaller trees for which it would be helpful to know the arity (branching factor) of? Which smaller trees are they? Write an example of a function call on one of these smaller trees. You can access these trees through the variable t.



3. Suppose the call in the previous step gives you the correct answer according to the docstring: it returns the arity of the tree it is given. How will you combine the solutions to all the smaller instances to get a solution for Tree t itself? Write code to return the correct thing. Put this code in the `else...` expression that you created in the fi step.

Go over these three parts with your TA. After that, fill in the implementation in `tree.py`, and see whether it works.

additional exercises

Here are some additional exercises on possibly_nested lists and some additional exercises on trees to try out if you finish the lab early.

The last 15 minutes of the lab will consist of a 15-minute quiz.