

Lecture12 Trees

Box-and-Pointer Notation

The Closure Property of Data Types



- A method for combining data values satisfies the *closure property* if:
 - The result of combination can itself be combined using the same method
- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

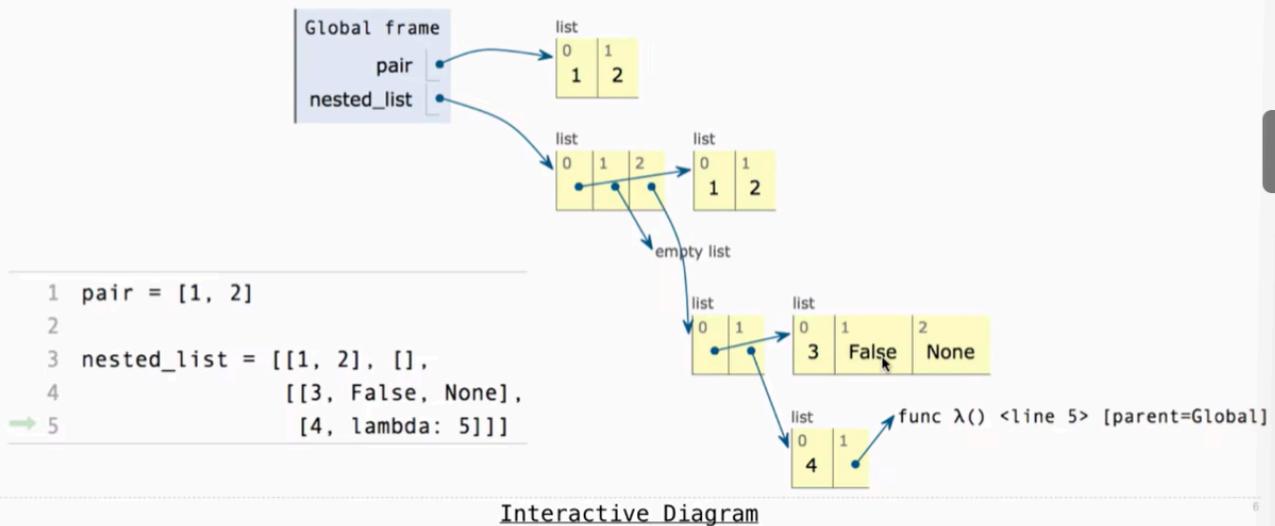
Lists can contain lists as elements (in addition to anything else)

Box-and-Pointer Notation in Environment Diagrams



Lists are represented as a row of index-labeled adjacent boxes, one per element

Each box either contains a primitive value or points to a compound value



Slices

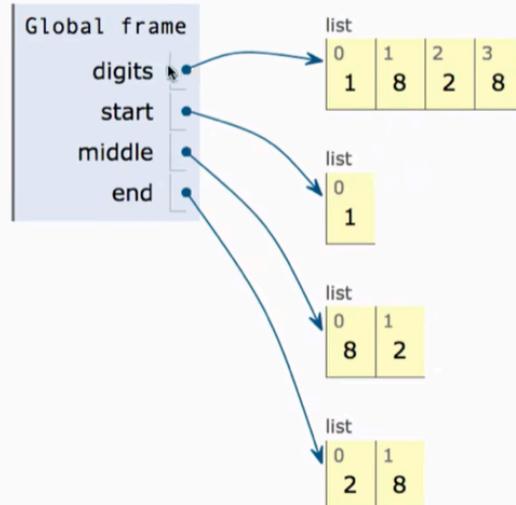
```
>>> odds = [3, 5, 7, 9, 11]
>>> list(range(1, 3))
[1, 2]
>>> [odds[i] for i in range(1, 3)]
[5, 7]
>>> odds[1:3]
[5, 7]
>>> odds[:3]
[3, 5, 7]
>>> odds[1:]
[5, 7, 9, 11]
>>> odds[:]
[3, 5, 7, 9, 11]
>>>
```



Slicing Creates New Values



```
1 digits = [1, 8, 2, 8]
2 start = digits[:1]
3 middle = digits[1:3]
4 end = digits[2:]
```



Processing Container Elements

Several built-in functions take iterable arguments and aggregate them into a value

```
sum(iterable[, start]) -> value
```

```
>>> sum([2, 3, 4])
```

```
9
```

```
>>> sum([2, 3, 4], 5)
```

```
14
```

```
# the start value is kinda like telling the com what is the expected value of  
this expression
```

```
max(iterable[, key = func]) -> value
```

```
max(a, b, c, ...[, key = func]) -> value
```

```
>>> max(range(4))
```

```
4
```

```

>>> max(0, 1, 2, 3, 4)
4

>>> max(range(10), lambda x: 7 - (x - 4) * (x - 2))
3
# the largest output from the result of the func taking the former list as
input

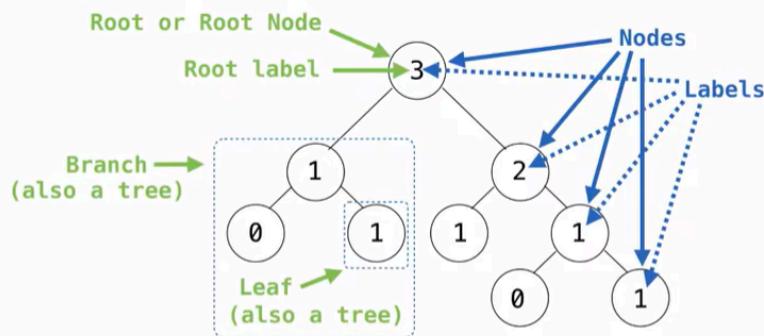
```

- `all(iterable) -> bool`

Return True if `bool(x)` is True for all values `x` in the iterable.
If the iterable is empty, return True.

Trees

Tree Abstraction



Recursive description (wooden trees):

A **tree** has a root **label** and a list of **branches**
Each branch is a **tree**
A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**
Each **node** has a **label** that can be any value
One node can be the **parent/child** of another



12

implementing the tree abstraction

Implementing the Tree Abstraction



```

def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be a tree'
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
  
```

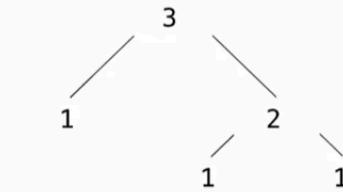
Verifies the tree definition

Creates a list from a sequence of branches

Verifies that tree is bound to a list

>>> tree(3, [tree(1),
... tree(2, [tree(1),
... tree(1)])])
[3, [1], [2, [1], [1]]]

def is_leaf(tree):
 return not branches(tree)



<

14

Tree Processing

```

~/lec$ python3 -i ex.py
>>> fib_tree(1)
[1]
>>> fib_tree(0)
[0]
>>> fib_tree(2)
[1, [0], [1]]
>>> fib_tree(4)
[3, [1, [0], [1]], [2, [1], [1, [0], [1]]]]
>>> label(fib_tree(4))
3
>>> 
  
```

Trees

```

def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

def is_leaf(tree):
    return not branches(tree)

def fib_tree(n):
    if n <= 1:
        return tree(n)
    else:
        left, right = fib_tree(n-2), fib_tree(n-1)
        return tree(label(left)+label(right), [left, right])
  
```

A small video window showing a man speaking.

Tree Processing Uses Recursion



Processing a leaf is often the base case of a tree processing function

The recursive case typically makes a recursive call on each branch, then aggregates

```
def count_leaves(t):
    """Count the leaves of a tree."""
    if is_leaf(t):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in branches(t)]
        return sum(branch_counts)
```

16

Discussion Question



Implement `leaves`, which returns a list of the leaf labels of a tree

Hint: If you `sum` a list of lists, you get a list containing the elements of those lists

```
>>> sum([ [1], [2, 3], [4] ], [])
[1, 2, 3, 4]
>>> sum([ [1] ], [])
[1]
>>> sum([ [[1]], [2] ], [])
[[1], 2]
def leaves(tree):
    """Return a list containing the leaf labels of tree.
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum(List of leaf labels for each branch, [])
branches(tree)
[ b for b in branches(tree) ]
leaves(tree)
[ s for s in leaves(tree) ]
[branches(b) for b in branches(tree)]
[leaves(b) for b in branches(tree)]
```

17

Creating Trees

Creating Trees



A function that creates a tree from another tree is typically also recursive

```
def increment_leaves(t):
    """Return a tree like t but with leaf labels incremented."""
    if is_leaf(t):
        return tree(label(t) + 1)
    else:
        bs = [increment_leaves(b) for b in branches(t)]
        return tree(label(t), bs)

def increment(t):
    """Return a tree like t but with all labels incremented."""
    return tree(label(t) + 1, [increment(b) for b in branches(t)])
```

18

Example

```
~/lec$ python3 -i ex.py
>>> print_tree(fib_tree(4))
3
1
0
1
2
1
1
0
1
>>> print_tree(fib_tree(5))
5
2
1
1
0
1
3
1
0
1
2
1
1
0
1
>>>
```

```
def fib_tree(n):
    if n <= 1:
        return tree(n)
    else:
        left, right = fib_tree(n-2), fib_tree(n-1)
        return tree(label(left)+label(right), [left, right])

def count_leaves(t):
    """Count the leaves of tree T."""
    if is_leaf(t):
        return 1
    else:
        return sum([count_leaves(b) for b in branches(t)])

def print_tree(t, indent=0):
    print(' ' * indent + str(label(t)))
    for b in branches(t):
        print_tree(b, indent+1)
```

A small portrait of a man with short brown hair, wearing a light-colored plaid shirt, positioned in the top right corner of the slide.