爱扑bug的熊

SICP

God's Programming Book

Lecture-11 Trees

Trees

Slides Adapted from cs61a of UC Berkeley

CC BY NC SA

Box-and-Pointer Notation

BY NC SA

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

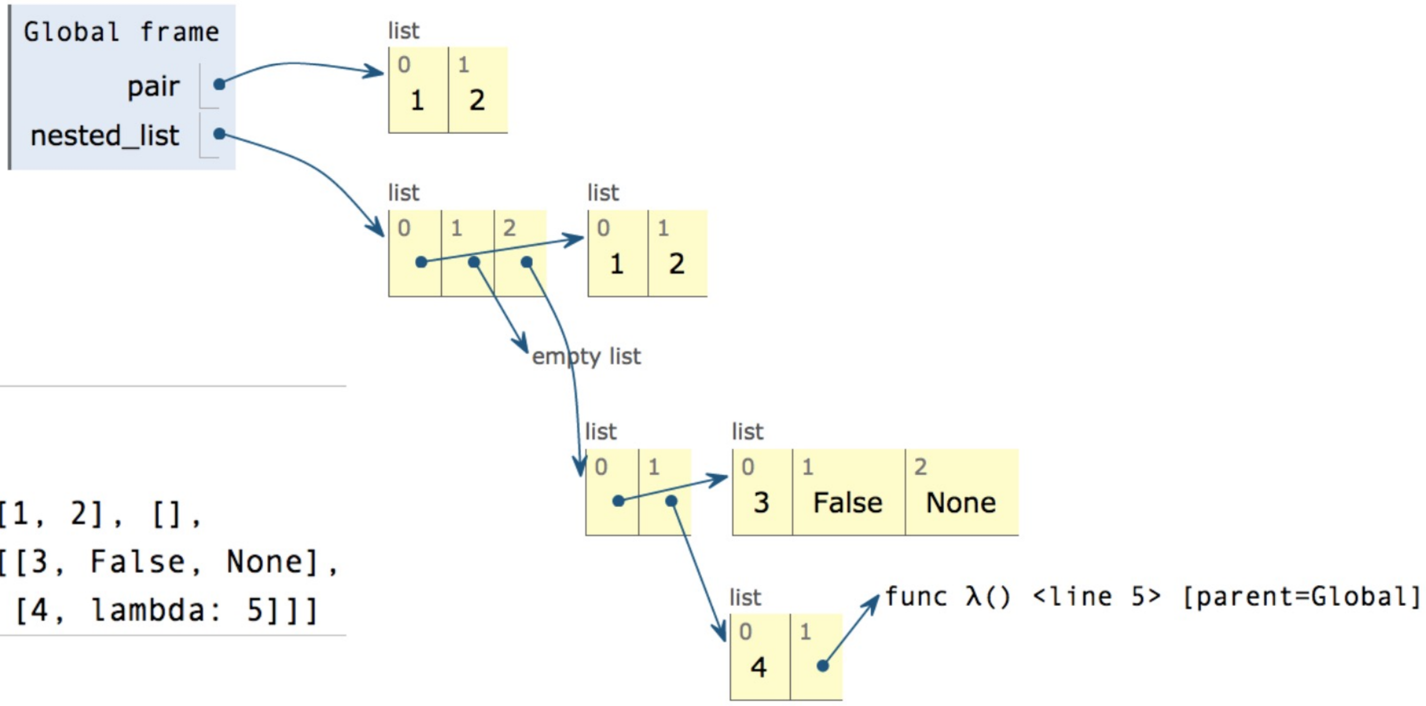
Lists can contain lists as elements (in addition to anything else)

CC BY NC SA

爱扑bug的熊

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




```
pair = [1, 2]

nested_list = [[1, 2], [],
               [[3, False, None],
                [4, lambda: 5]]]
```

Slicing

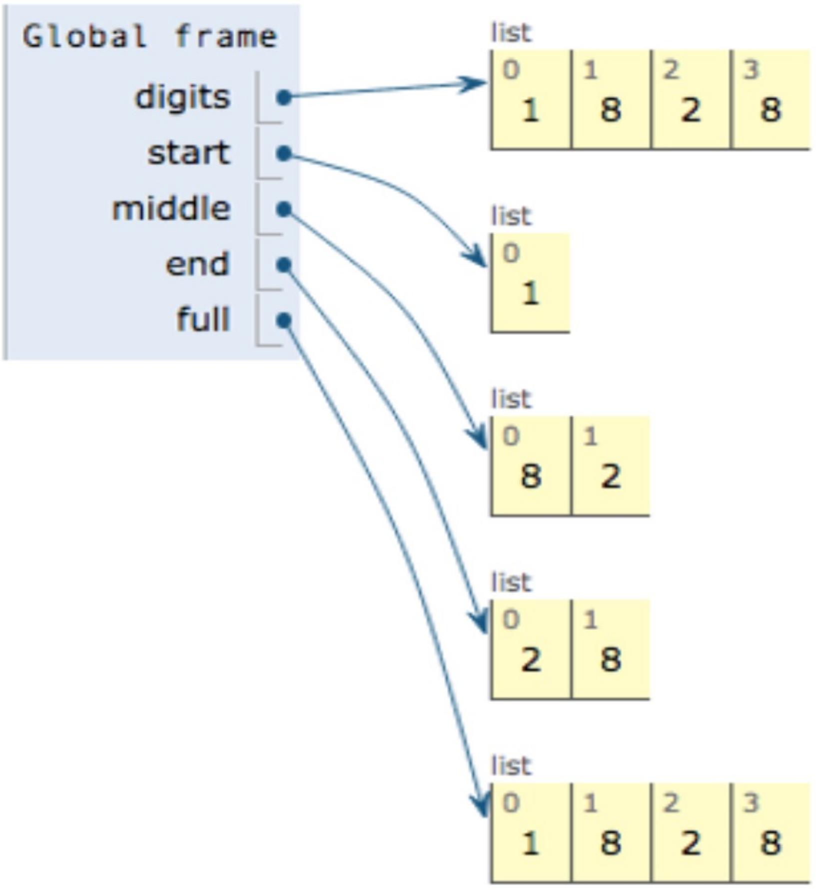
(Demo)

CC BY NC SA

爱扑bug的熊

Slicing Creates New Values

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



Processing Container Values

cc BY NC SA

Sequence Aggregation

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

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

  Return the sum of a 'start' value (default: 0) plus an iterable of numbers.

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

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

  With a single iterable argument, return its largest item.

  With two or more arguments, return the largest argument.

- all(iterable) -> bool

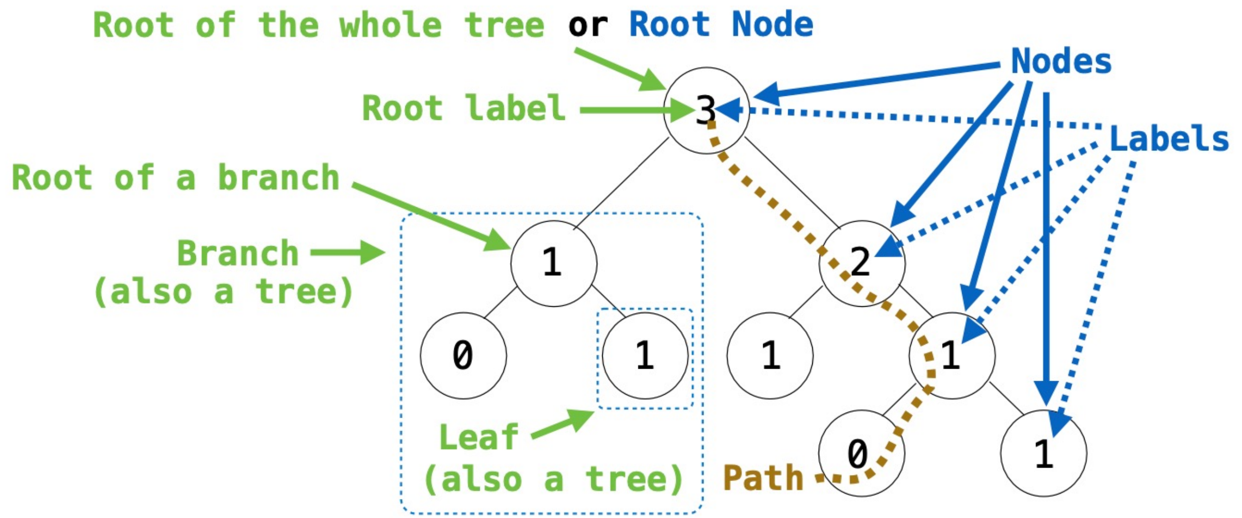
  Return True if bool(x) is True for all values x in the iterable. If the iterable is empty, return True.

Trees

BY NC SA

爱扑bug的熊

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**

A **tree** starts at the **root**

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

The top node is the **root node**

People often refer to labels by their locations: "each parent is the sum of its children"

爱扑bug的熊

Implementing the Tree Abstraction

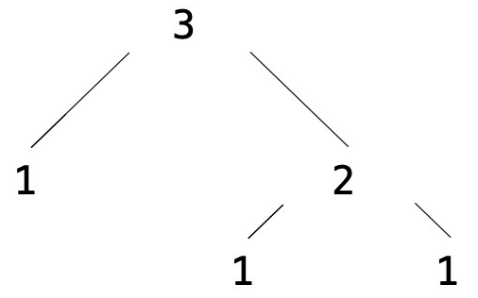
```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

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

- A **tree** has a root **label** and a list of **branches**
- Each branch is a tree




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

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

CC BY NC SA

Tree Processing

CC BY NC SA

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)
```

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)]   [branches(s) for s in leaves(tree)]

[leaves(b) for b in branches(tree)]     [leaves(s) for s in leaves(tree)]
```

CC BY NC SA

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)])
```

CC BY NC SA

Example: Printing Trees

(Demo)

CC BY NC SA

Example: Summing Paths

(Demo)

CC BY NC SA

Thanks for Listening