CS 61A Lecture 11
Box-and-Pointer Notation
The Closure Property of Data Types
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
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
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
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

Interactive Diagram
Box-and-Pointer Notation in Environment Diagrams

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

```python
pair = [1, 2]
```
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]
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.

```
1  pair = [1, 2]
2
3  nested_list = [[1, 2], [],
4     [[3, False, None],
5     [4, lambda: 5]]]
```
Sequence Operations
Membership & Slicing

Python sequences have operators for membership and slicing
Membership & Slicing

Python sequences have operators for membership and slicing

Membership.
Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```python
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```
Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```python
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.
Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```python
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```python
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```
Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```python
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```python
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object
Membership & Slicing

Python sequences have operators for membership and slicing

### Membership.

```python
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

### Slicing.

```python
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object.
Membership & Slicing

Python sequences have operators for membership and slicing:

Membership.

```python
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```python
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object.
Trees
Tree Abstraction
Tree Abstraction
A tree has a root value and a sequence of branches; each branch is a tree.
A tree has a root value and a sequence of branches; each branch is a tree.
A tree has a root value and a sequence of branches; each branch is a tree.
A tree has a root value and a sequence of branches; each branch is a tree.

A tree with zero branches is called a leaf.
A tree has a root value and a sequence of branches; each branch is a tree.

A tree with zero branches is called a leaf.
A tree has a root value and a sequence of branches; each branch is a tree.

A tree with zero branches is called a leaf.
Tree Abstraction

A tree has a root value and a sequence of branches; each branch is a tree.

A tree with zero branches is called a leaf.

The root values of sub-trees within a tree are often called node values or nodes.
A tree has a root value and a sequence of branches; each branch is a tree.

A tree with zero branches is called a leaf.

The root values of sub-trees within a tree are often called node values or nodes.
A tree has a root value and a sequence of branches; each branch is a tree.

A tree with zero branches is called a leaf.

The root values of sub-trees within a tree are often called node values or nodes.
Implementing the Tree Abstraction
Implementing the Tree Abstraction

A tree has a root value and a sequence of branches; each branch is a tree.
Implementing the Tree Abstraction

A tree has a root value and a sequence of branches; each branch is a tree.
Implementing the Tree Abstraction

A tree has a root value and a sequence of branches; each branch is a tree

```
>>> tree(3, [tree(1),
          ... tree(2, [tree(1),
          ... tree(1)]))])
```
Implementing the Tree Abstraction

A tree has a root value and a sequence of branches; each branch is a tree.

```
>>> tree(3, [tree(1),
    ... tree(2, [tree(1),
    ... tree(1)]),
    ... [3, [1], [2, [1], [1]]])
```
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
```

A tree has a root value and a sequence of branches; each branch is a tree

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

[3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
    return [root] + branches
```

A tree has a root value and a sequence of branches; each branch is a tree.

```python
>>> tree(3, [tree(1),
...    tree(2, [tree(1),
...    tree(1)]),
...    [3, [1], [2, [1], [1]]])
```
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
    return [root] + branches

def root(tree):
>>> tree(3, [tree(1),
...   tree(2, [tree(1),
...     tree(1)]),
...   [3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
    return [root] + branches

def root(tree):
    return tree[0]

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

A tree has a root value and a sequence of branches; each branch is a tree.
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
    return [root] + branches

def root(tree):
    return tree[0]

def branches(tree):
```

A tree has a root value and a sequence of branches; each branch is a tree.
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
    return [root] + branches

def root(tree):
    return tree[0]

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

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

A tree has a root value and a sequence of branches; each branch is a tree.
Implementing the Tree Abstraction

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

def root(tree):
    return tree[0]

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

>>> tree(3, [tree(1),
...    tree(2, [tree(1),
...      tree(1)])]),
...    [3, [1], [2, [1], [1]]])
```
Implementing the Tree Abstraction

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

def root(tree):
    return tree[0]

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

>>> tree(3, [tree(1),
         ...   tree(2, [tree(1),
         ...     tree(1)]),
         ... [3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

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

def root(tree):
    return tree[0]

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

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

A tree has a root value and a sequence of branches; each branch is a tree
```
Implementing the Tree Abstraction

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

def root(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
```
Implementing the Tree Abstraction

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

def root(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
```

A tree has a root value and a sequence of branches; each branch is a tree.
Implementing the Tree Abstraction

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

A tree has a root value and a sequence of branches; each branch is a tree.

```

```python
def root(tree):
    return tree[0]

Creates a list from a sequence of branches.

```

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

Verifies that tree is bound to a list.

```

```python
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

```

```python
def tree(root, branches=[]):
    # Verifies the tree definition

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

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

```

```python
def root(tree):
    return tree[0]

```

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

```

```python
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

```

```python
def tree(root, branches=[]):
    # Verifies the tree definition
```

```python
def root(tree):
    return tree[0]

Creates a list from a sequence of branches.

```

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

Verifies that tree is bound to a list.

```python
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

```

```python
def tree(root, branches=[]):
    # Verifies the tree definition

>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                 tree(1)]),
...                 [3, [1], [2, [1], [1]]])]
```
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
    # Verifies the tree definition
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)

def root(tree):
    # Creates a list from a sequence of branches
    return tree[0]

def branches(tree):
    # Verifies that tree is bound to a list
    return tree[1:]

def is_tree(tree):
    # Verifies the tree definition
    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 tree(root, branches=[]):
    # Verifies the tree definition
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)

def root(tree):
    # Creates a list from a sequence of branches
    return tree[0]

def branches(tree):
    # Verifies that tree is bound to a list
    return tree[1:]

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

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

def is_leaf(tree):
    return not branches(tree)  # Demo
```
Tree Processing
Tree Processing Uses Recursion
def count_leaves(tree):
    """Count the leaves of a tree."""
Tree Processing Uses Recursion

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

```python
def count_leaves(tree):
    """Count the leaves of a tree."""
```
Tree Processing Uses Recursion

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

```python
def count_leaves(tree):
    """Count the leaves of a tree."""
    if is_leaf(tree):
        return 1
```
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.

```python
def count_leaves(tree):
    """Count the leaves of a tree."""
    if is_leaf(tree):
        return 1
```
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.

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

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

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

(Demo)
Discussion Question
Discussion Question

Implement *leaves*, which returns a list of the leaf values of a tree
**Discussion Question**

Implement \texttt{leaves}, which returns a list of the leaf values of a tree

```python
def leaves(tree):
    """Return a list containing the leaves of tree."

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
```

Discussion Question

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

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

```python
def leaves(tree):
    """Return a list containing the leaves of tree."

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
```

---

15
**Discussion Question**

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

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

```python
>>> sum([ [1], [2, 3], [4] ], [])
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
```

```
Discussion Question

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

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

```python
>>> sum([[1], [2, 3], [4], []])
[1, 2, 3, 4]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
```
Discussion Question

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

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

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
```

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]

>>> sum([[1]], [])
[1]
```
Discussion Question

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

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

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1]], [])
[1]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
```
Discussion Question

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

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

```python
>>> sum([ [1], [2, 3], [4] ], [])
[1, 2, 3, 4]
>>> sum([ [1] ], [])
[1]
>>> sum([ [[1]], [2] ], [])
[1, 0, 1, 0, 1, 1, 0, 1]
```

def leaves(tree):
    """Return a list containing the leaves of tree."""
    return [x for x in tree if not isinstance(x, list)]
Discussion Question

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

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

```python
def leaves(tree):
    """Return a list containing the leaves of tree."""
    return sum(tree, [])

>>> leaves(fib_tree(5))
[1, 0, 1, 0, 1, 1, 0, 1]
```

```python
def fib_tree(n):
    if n == 0:
        return [1]
    leaf = fib_tree(n-1)
    return [sum(leaf), sum(leaf[1:])] + fib_tree(n-1)
```
Discussion Question

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

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

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
```

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1]], [])
[1]
>>> sum([[1]], [2], [])
[[1], 2]
```
Discussion Question

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

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

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1], []])
[1]
>>> sum([[1]], [2], [])
[[1], 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum(______________________________, [])

>>> leaves(fib_tree(5))
[1, 0, 1, 0, 1, 1, 0, 1]"""
Discussion Question

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

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

```python
def leaves(tree):
    """Return a list containing the leaves of tree."""
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum(branches(tree), [])

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

```python
>>> leaves(fib_tree(5))
[1, 0, 1, 0, 1, 1, 0, 1]
```

```python
>>> sum([1, 2, 3, 4])
[1, 2, 3, 4]
>>> sum([[1]], [])
[1]
>>> sum([[1], [2]], [])
[[1], 2]
```
Discussion Question

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

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

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum([leaves(b) for b in branches(tree)], [])
```

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

```python
[1, 0, 1, 0, 1, 1, 0, 1]
```
Discussion Question

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

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

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1], []])
[1]
>>> sum([[1]], [2], [])
[[1], 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."

    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum(List of leaves for each branch, [])
```

```python
branches(tree)
leaves(tree)
[branches(b) for b in branches(tree)]
[leaves(b) for b in branches(tree)]
```
Example: Partition Trees

(Demo)

Interactive Diagram