CS 61A Lecture 11

Wednesday, February 18
Announcements
Announcements

• Optional Hog Contest due Wednesday 2/18 @ 11:59pm
Announcements

• Optional Hog Contest due Wednesday 2/18 @ 11:59pm

• Homework 3 due Thursday 2/19 @ 11:59pm
Announcements

• Optional Hog Contest due Wednesday 2/18 @ 11:59pm

• Homework 3 due Thursday 2/19 @ 11:59pm

• Project 2 due Thursday 2/26 @ 11:59pm
Announcements

• Optional Hog Contest due Wednesday 2/18 @ 11:59pm

• Homework 3 due Thursday 2/19 @ 11:59pm

• Project 2 due Thursday 2/26 @ 11:59pm
  ▪ Bonus point for early submission by Wednesday 2/25 @ 11:59pm!
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
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.

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], [],
                  [[3, False, None],
                   [4, lambda: 5]]]
```

Interactive Diagram
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]
```
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.
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.
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.

```python
>>> 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

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

def tree(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

>>> 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

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

def root(tree):
```

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

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

def root(tree):
    return tree[0]

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

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.

```
>>> 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]

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]]]
```

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]]]
```

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:]
```

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]])
...
[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:]

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.

```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=[]):
    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

>>> 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

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

Verifies that tree is bound to a list

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

Verifies the tree definition

def is_leaf(tree):
    return not branches(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

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

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

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 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 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 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 `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]
```

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]], [])
[1]
```

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], []], [])
[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
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], []])
[1]
>>> sum([[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], []])
[1]
>>> sum([[1], [2]], [])
[[1], 2]
```

```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]], [])
[1]
>>> sum([[1]], [2], [])
[1, 2]
>>> 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
>>> sum([[1]], [])
[1]
>>> sum([[1], [2]], [])
[1, 2]
>>> 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 ________________________________
```
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([leaves(b) for b in branches(tree)], [])

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

```python
>>> sum([[1], []])
[1]
>>> sum([[1], [2]], [])
[1, 2]
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
```
Example: Partition Trees

(Demo)

Interactive Diagram