Trees
Announcements
Congratulations to the Winners of the Hog Strategy Contest
Congratulations to the Winners of the Hog Strategy Contest

1st Place with 146 wins:
Congratulations to the Winners of the Hog Strategy Contest

1st Place with 146 wins:

A five-way tie for first place!
Congratulations to the Winners of the Hog Strategy Contest

1st Place with 146 wins:

A five-way tie for first place!

"A submission scores a match point each time it has an expected win rate strictly above 50.0001%."
Congratulations to the Winners of the Hog Strategy Contest

1st Place with 146 wins:

A five-way tie for first place!

"A submission scores a match point each time it has an expected win rate strictly above 50.0001%."

Bobby Tables

blockchain

Anonymous Poet

wet app program

1.6180339887
Congratulations to the Winners of the Hog Strategy Contest

1st Place with 146 wins:

A five-way tie for first place!

"A submission scores a match point each time it has an expected win rate strictly above 50.0001%.

Congratulations to Timothy Guo, Shomini Sen, Samuel Berkun, Mitchell Zhen, Lucas Clark, Dominic de Bettencourt, Allen Gu, Alec Li, Aaron Janse
Congratulations to the Winners of the Hog Strategy Contest

1st Place with 146 wins:

A five-way tie for first place!

"A submission scores a match point each time it has an expected win rate strictly above 50.0001%.

Bobby Tables

blockchain

Anonymous Poet

wet app program

1.6180339887

Congratulations to Timothy Guo, Shomini Sen, Samuel Berkun, Mitchell Zhen, Lucas Clark, Dominic de Bettencourt, Allen Gu, Alec Li, Aaron Janse

hog-contest.cs61a.org
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  nested_list = [[1, 2], [],
3     [[3, False, None],
4     [4, lambda: 5]]]
Slicing

(Demo)
Slicing Creates New Values

```
1 digits = [1, 8, 2, 8]
2 start = digits[:1]
3 middle = digits[1:3]
4 end = digits[2:]
5 full = digits[:]
```
Processing Container Values
Sequence Aggregation
Sequence Aggregation

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

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

- \texttt{sum(iterable[, start])} \rightarrow \texttt{value}

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

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

- **sum(iterable[, start])** \(\rightarrow\) value
  
  Return the sum of a 'start' value (default: 0) plus an iterable of numbers.

- **max(iterable[, key=func])** \(\rightarrow\) value
  
  With a single iterable argument, return its largest item.
  With two or more arguments, return the largest argument.

- **all(iterable)** \(\rightarrow\) 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

```
    3
   /|
  /  \
1    2
 /    /
0   1   1
  \
   1   0
    \  
     1
```
Tree Abstraction

**Recursive description** (wooden trees):  

**Relative description** (family trees):
Tree Abstraction

Recursive description (wooden trees):
A tree has a root label and a list of branches

Relative description (family trees):
Tree Abstraction

**Recursive description** (wooden trees): A tree has a **root label** and a list of **branches**

**Relative description** (family trees):
Recursive description (wooden trees): A tree has a root label and a list of branches

Relative description (family trees):
Tree Abstraction

Recursive description (wooden trees):
A tree has a root label and a list of branches.
Each branch is a tree.

Relative description (family trees):
**Recursive description** (wooden trees): A tree has a root label and a list of branches. Each branch is a tree.

**Relative description** (family 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):**
- Root label (also a tree)

Diagram:
- Node 3 (root)
  - Branch 1
    - Node 0
    - Node 1
  - Branch 2
    - Node 1
    - Node 0
    - Node 1
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):**
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):
**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):**
A **root label**
A **branch** (also a tree)
A **leaf** (also a tree)
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):

A **root label**

**Root of the whole tree**

**Root of a branch**

**Branch** (also a tree)

**Leaf** (also a tree)
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.
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
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
**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
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**
**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**
**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.

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

**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.
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"
Implementing the Tree Abstraction
Implementing the Tree Abstraction

• A tree has a root label and a list of branches
• Each branch is a tree
Implementing the Tree Abstraction

- A tree has a root label and a list of branches.
- Each branch is a tree.
Implementing the Tree Abstraction

- 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)])])
```
Implementing the Tree Abstraction

- 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]]]
```
Implementing the Tree Abstraction

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

• 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]]]
```
Implementing the Tree Abstraction

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

• 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]]]
Implementing the Tree Abstraction

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

def label(tree):

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

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

```
3
  / \
 1   2
  /   /
/     / \
1     1 1
```

Implementing the Tree Abstraction

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

def label(tree):
    return tree[0]
```

- A tree has a root label and a list 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(label, branches=[]):
    return [label] + branches

def label(tree):
    return tree[0]

def branches(tree):
```

- 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]]])
```

```
3
  1
 / 
1 2
 / 
1 1
```
Implementing the Tree Abstraction

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

def label(tree):
    return tree[0]

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

- A tree has a root `label` and a list 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(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:]

• 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(2), [tree(1)]]))])
[3, [1], [2, [1], [1]]]
Implementing the Tree Abstraction

```python
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:]
```

- A tree has a root `label` and a list 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(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:]

Verifies the tree definition

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]]]
Implementing the Tree Abstraction

```python
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

- A tree has a root label and a list 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(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
```

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

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

```python
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
# - 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)
```
Implementing the Tree Abstraction

```python
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

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

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

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

```
        3
       / \  
      1   2
     /   /  \
    1   1   1
```

Verifies that tree is bound to a list

Creates a list from a sequence of branches

Verifies the tree definition
Tree Processing
Tree Processing

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

Tree Processing Uses Recursion
Tree Processing Uses Recursion

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

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

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

(Demo)
Discussion Question
Discussion Question

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

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

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

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

Discussion Question

Implement the function `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.

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

```python
def leaves(tree):
    """Return a list containing the leaf labels 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 labels 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 leaf labels 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 labels 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
>>> sum([[1]], [])
[1]
```

```python
def leaves(tree):
    """Return a list containing the leaf labels 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 labels 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 leaf labels of tree."
    return [label for sublist in tree for label in sublist if not sublist]

>>> leaves(fib_tree(5))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

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

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

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

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

```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 leaf labels 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 labels 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 leaf labels of tree."""
    return [x for x in tree if not isinstance(x, list)]

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

```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 leaf labels of tree."
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum(______________________________, [])
```
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

```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 leaf labels of tree."
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum(______________________________, [])
```

```python
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]
```
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

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

```
Discussion Question

Implement the `leaves` function, 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.

```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 leaf labels of tree."
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum([leaves(b) for b in branches(tree)], [])

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

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

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

```python
def increment_leaves(t):
    """Return a tree like t but with leaf labels incremented."""
```
Creating Trees

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

```python
def increment_leaves(t):
    """Return a tree like t but with leaf labels incremented.""
    if is_leaf(t):
        return tree(label(t) + 1)
```
A function that creates a tree from another tree is typically also recursive

```python
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)
```
Creating Trees

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

```python
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."""
```
Creating Trees

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

```python
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)])
```
Example: Printing Trees

(Demo)
Example: Summing Paths

(Demo)