Trees
Announcements
Hog Contest Winners

Strategy contest:  https://hog-contest.cs61a.org/

3-way tie for first: Nishant Bhakar, Toby Worledge, Asrith Devalaraju & Aayush Gupta

After bug fix: (1) Nishant Bhakar, (2) Toby Worledge, (3) Jiayin Lin & Roger Yu

Dice contest:  https://dice.cs61a.org/

(1) Bella Lee & Dayeon Jang  (2) Michelle Wu & Kevin Xu  (3) Taylor Moore
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.

```
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]]]
```

```
Global frame
  pair
  nested_list

list 0 1 2
list 0 1 2
list 0 1 2
empty list
list 0 1
list 0 1 2 3 4

func λ() <line 5> [parent=Global]
```
Slicing

(Demo)
Slicing Creates New Values

```python
digits = [1, 8, 2, 8]
start = digits[1:2]
middle = digits[1:3]
end = digits[2:]
full = digits[:]
```
Processing Container Values
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
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

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

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

```python
def tree(label, branches=[]):
    # Example usage
    >>> tree(3, [tree(1),
                 ...    tree(2, [tree(1),
                 ...       [3, [1], [2, [1], [1]]])])

    [3, [1], [2, [1], [1]]]
```
Tree Processing

(Demo)
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)
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
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)
Example: Counting Paths
Count Paths that have a Total Label Sum

def count_paths(t, total):
    """Return the number of paths from the root to any node in tree t
    for which the labels along the path sum to total.
   "

    if label(t) == total:
        found = 1
    else:
        found = 0

    return found + sum([_count_paths(b, total - label(t)) for b in branches(t)])