CS 61A Lecture 11

Announcements

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

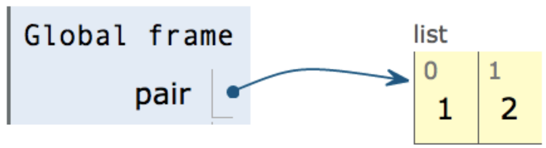
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

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]
```

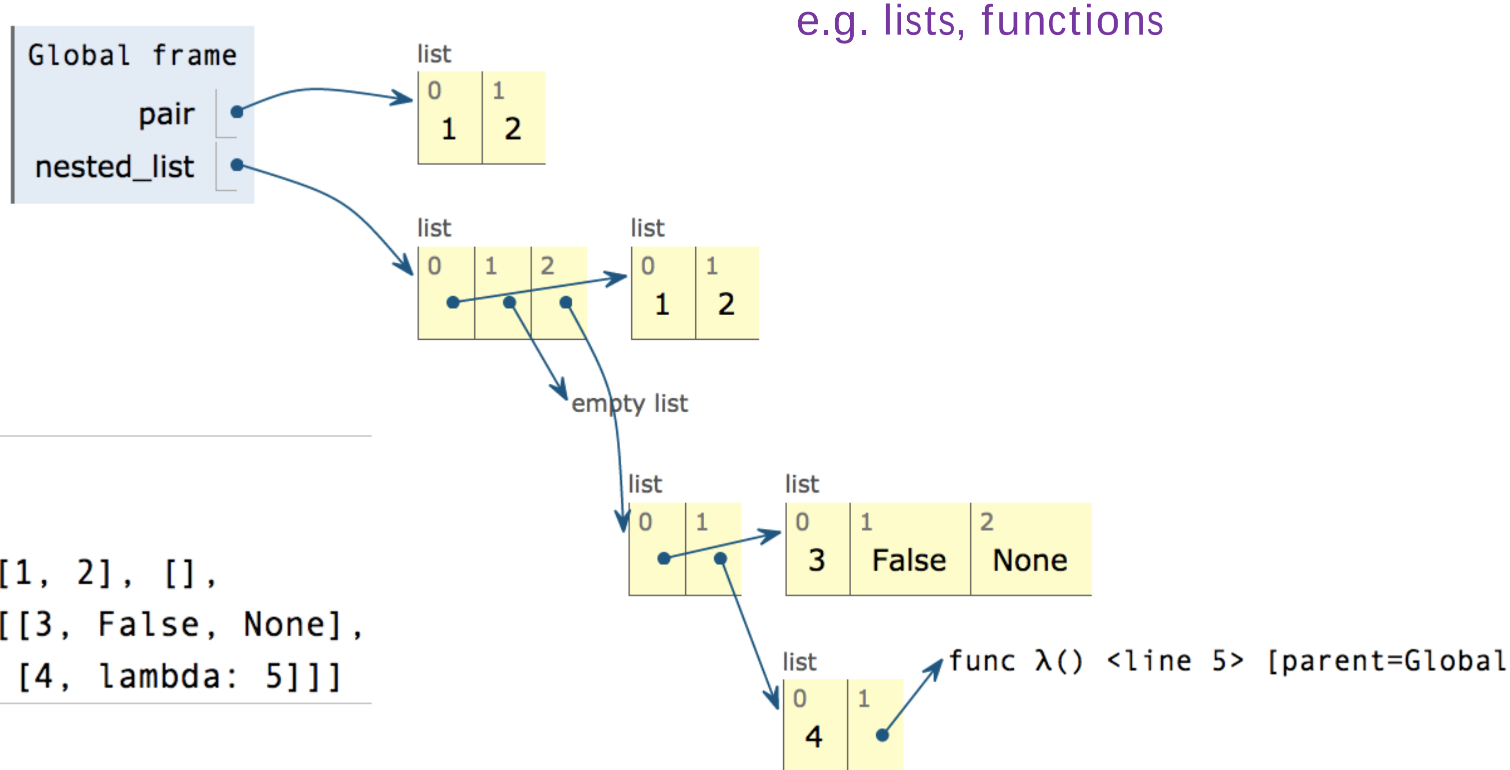
Interactive Diagram

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

e.g. lists, functions




```
pair = [1, 2]

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

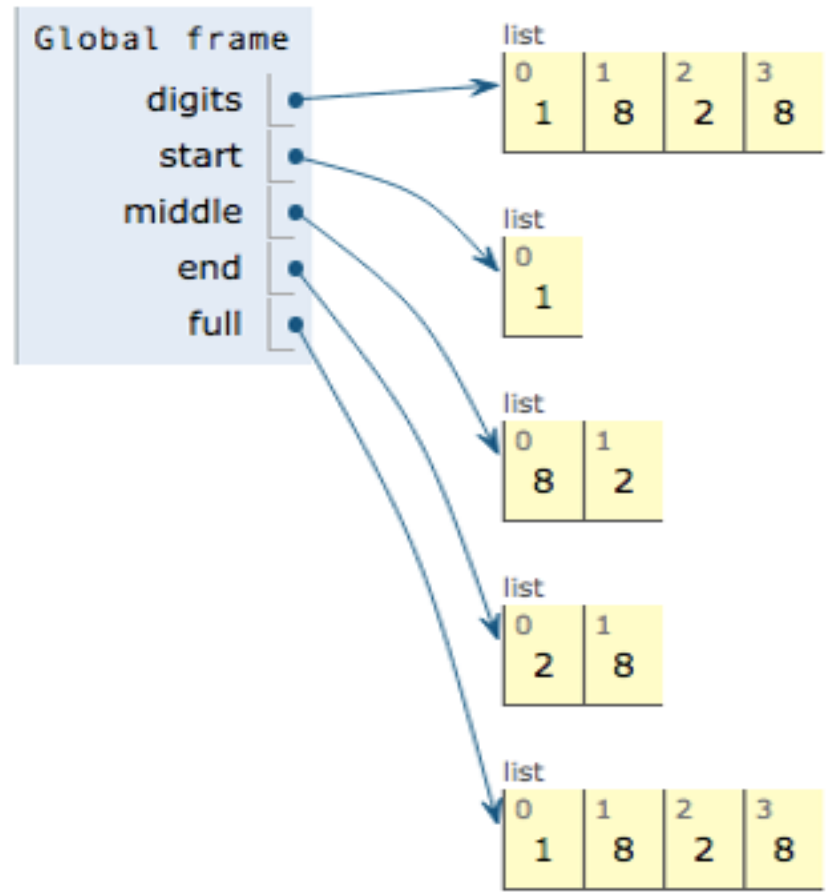
Interactive Diagram

Slicing

(Demo)

Slicing Creates New Values

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



Interactive Diagram

Processing Container Values

Sequence Aggregation

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

optional

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

  Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

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

  also min

  With a single iterable argument, return its largest item.
  With two or more arguments, return the largest argument.

- **all**(iterable) -> bool

  also any

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

Sequence Aggregation

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

```
>>> sum([2, 3, 4])
9
>>> sum(['2', '3', '4'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> sum([2, 3, 4], 5)
14
>>> [2, 3] + [4]
[2, 3, 4]
>>> sum([[2, 3], [4]], [])
[2, 3, 4]
>>> 0 + [2, 3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>> sum([[2, 3], [4]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>> [] + [2, 3] + [4]
[2, 3, 4]
```

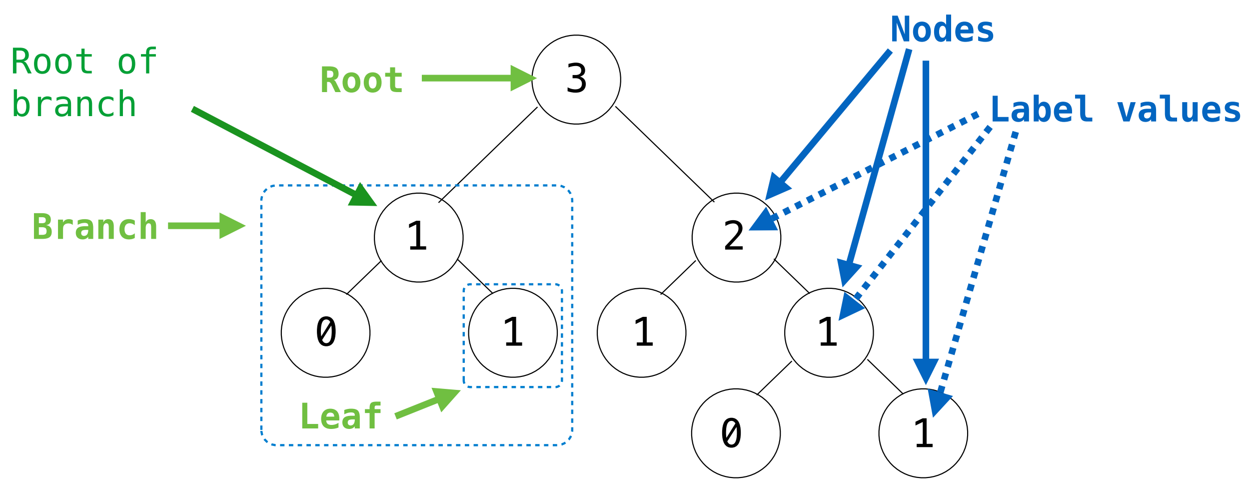
the usage of start: add lists in a list

```
>>> max(range(5))
4
>>> max(0, 1, 2, 3, 4)
4
>>> max(range(10), key=lambda x: 7-(x-4)*(x-2))
3
```

```
>>> range(5)
range(0, 5)
>>> [x < 5 for x in range(5)]
[True, True, True, True, True]
>>> all([x < 5 for x in range(5)])
True
>>> all(range(5))
False
```

Trees

Tree Abstraction



Recursive description (wooden trees):

A **tree** has a **root** and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label value**

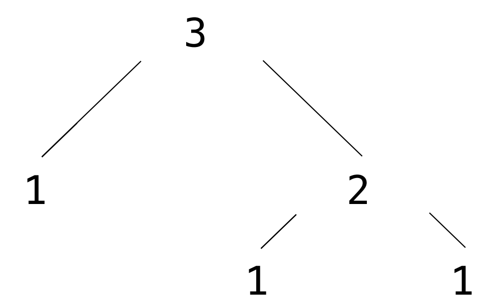
One node can be the **parent/child** of another

People often refer to values 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:]
```

- A tree has a label value and a list of branches




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

Implementing the Tree Abstraction

tree constructor

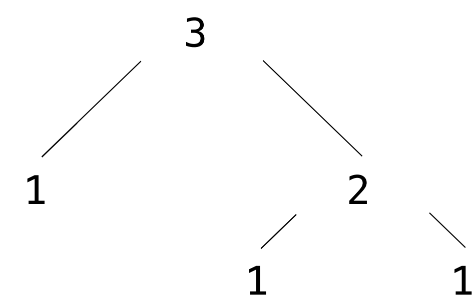
```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

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 label value and a list of branches




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

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

(Demo)

Tree Processing

(Demo)

more often we construct trees using fun def instead of tree constructor:

```
def fib_tree(n):
    if n <= 1:
        return tree(n)
    else:
        left, right = fib_tree(n-2), fib_tree(n-1)
        return tree(label(left)+label(right), [left, right])
```

```
>>> fib_tree(2)
[1, [0], [1]]
>>> fib_tree(4)
[3, [1, [0], [1]], [2, [1], [1, [0], [1]]]]
>>> label(fib_tree(4))
3
```

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)
```

(Demo)

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 leaves 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 leaves for each branch, []))
```

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

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

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 values 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 node values incremented."""
    return tree(label(t) + 1, [increment(b) for b in branches(t)])
```

a trick: when base case, the second argument -> []

```python
def print_tree(t, indent=0):
    print('  ' * indent + str(label(t)))
    for b in branches(t):
        print_tree(b, indent+1)
```

Example: Printing Trees

```
>>> print_tree(fib_tree(5))
5
  2
    1
    1
      0
      1
  3
    1
      0
      1
    2
      1
      1
        0
        1
```

(Demo)