Lecture #10: Sequences to Trees

Review: Sequence Comprehension

- Syntax:

```
[ <expr> for <var> in <sequence expr> ]
[ <expr> for <var> in <sequence expr> if <boolean expression> ]
```

- Examples:

```
>>> [ 2**x for x in range(5) ]
[1, 2, 4, 8, 16 ]
>>> L = [5, 7, 8, 10, 6, 8, 7, 4, 9, 8]
>>> [ x for x in L if x % 2 == 1 ]
[ 5, 7, 7, 9 ]
```

- In fact, the syntax is more general:

```
>>> [ (x, y) for x in range(2) for y in range(3) ]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
>>> # Still one-dimensional; y varies fastest
```

Representing Multi-Dimensional Structures

- How do we represent a two-dimensional table (like a matrix)?
- Answer: use a *sequence of sequences* (typically a list of lists or tuple of tuples).
- The same approach is used in C, C++, and Java.
- Example:

$$\begin{bmatrix} 1 & 2 & 0 & 4 \\ 0 & 1 & 3 & -1 \\ 0 & 0 & 1 & 8 \end{bmatrix}$$

becomes

```
(( 1, 2, 0, 4 ), ( 0, 1, 3, -1), (0, 0, 1, 8))
     # or
[[ 1, 2, 0, 4 ], [ 0, 1, 3, -1], [0, 0, 1, 8]]
     # or (for old Fortran hands):
[[ 1, 0, 0 ], [ 2, 1, 0 ], [ 0, 3, 1 ], [ 4, -1, 8 ]]
```

Problem: Creating A Two-Dimensional Table

```
def multiplication_table(rows, cols):
    """A ROWS x COLS multiplication table where row x, column y
    (element [x][y]) contains xy.  Example:
    >>> multiplication_table(4, 3)
    [[0, 0, 0], [0, 1, 2], [0, 2, 4], [0, 3, 6]]
    """
    return ________________________________________
```

Problem: Creating A Two-Dimensional Table (II)

```
def multiplication_table(rows, cols):
    """A ROWS x COLS multiplication table where row x, column y
    (element [x][y]) contains xy.  Example:
    >>> multiplication_table(4, 3)
    [[0, 0, 0], [0, 1, 2], [0, 2, 4], [0, 3, 6]]
    """
    return [ _________________________________
             for row in range(rows) ]
```

Problem: Creating A Two-Dimensional Table (III)

```
def multiplication_table(rows, cols):
    """A ROWS x COLS multiplication table where row x, column y
    (element [x][y]) contains xy.  Example:
    >>> multiplication_table(4, 3)
    [[0, 0, 0], [0, 1, 2], [0, 2, 4], [0, 3, 6]]
    """
    return [ [ row * col for col in range(cols) ]
             for row in range(rows) ]
```

Problem: Creating a Triangular Array

- There's no reason the rows in a 2D list must have the same length.

```
def triangle(rows):
    """A ROWSxROWS lower-triangular array
    containing "*"s.
    >>> triangle(4)
    [['*'], ['*', '*'], ['*', '*', '*'], ['*', '*', '*', '*']]
    """
```

Problem: Creating a Triangular Array (II)

- There's no reason the rows in a 2D list must have the same length.

```
def triangle(rows):
    """A ROWSxROWS lower-triangular array
    containing "*"s.
    >>> triangle(4)
    [['*'], ['*', '*'], ['*', '*', '*'], ['*', '*', '*', '*']]
    """
    return [ [ "*" for c in range(k+1) ] for k in range(rows) ]
```

Variation: Creating a Numbered Triangular Array

- This time, use numbers instead of asterisks.

```
def numbered_triangle(rows):
    """A ROWSxROWS lower-triangular array whose elements
    are integers, starting at 0 going left-to-right,
    up-to-down.
    >>> numbered_triangle(3)
    [ [ 0 ], [ 1, 2 ], [ 3, 4, 5 ] ]"""
```

Creating a Numbered Triangular Array (II)

- This time, use numbers instead of asterisks.

```
def numbered_triangle(rows):
    """A ROWSxROWS lower-triangular array whose elements
    are integers, starting at 0 going left-to-right,
    up-to-down.
    >>> numbered_triangle(3)
    [ [ 0 ], [ 1, 2 ], [ 3, 4, 5 ] ]"""
    def first(row):
        """The ROWth triangular number."""
        return (row * row + row) // 2
    return ________________________________
```

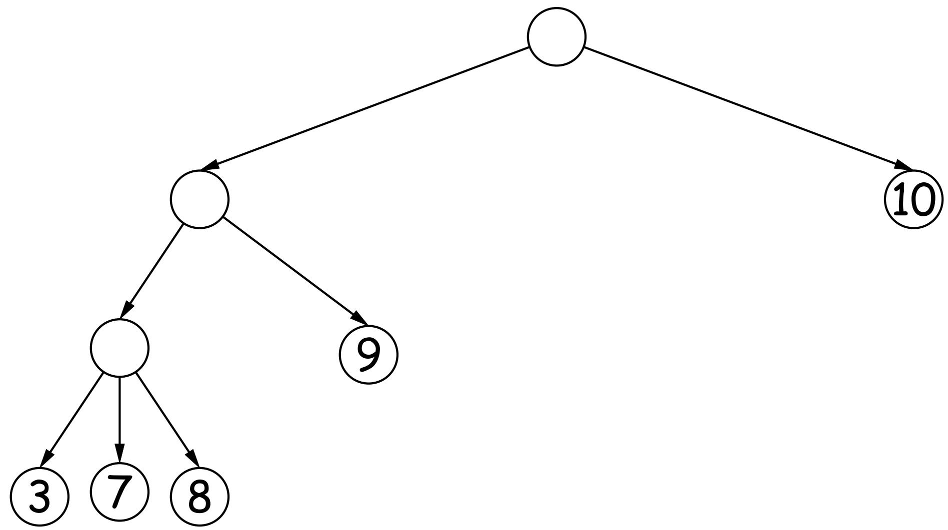
Creating a Numbered Triangular Array (III)

- This time, use numbers instead of asterisks.

```
def numbered_triangle(rows):
    """A ROWSxROWS lower-triangular array whose elements
    are integers, starting at 0 going left-to-right,
    up-to-down.
    >>> numbered_triangle(3)
    [ [ 0 ], [ 1, 2 ], [ 3, 4, 5 ] ]"""
    def first(row):
        """The ROWth triangular number."""
        return (row * row + row) // 2
    return [ [ x for x in range(first(row), first(row) + row + 1) ]
             for row in range(rows) ]
```

And Why Stop There? Trees

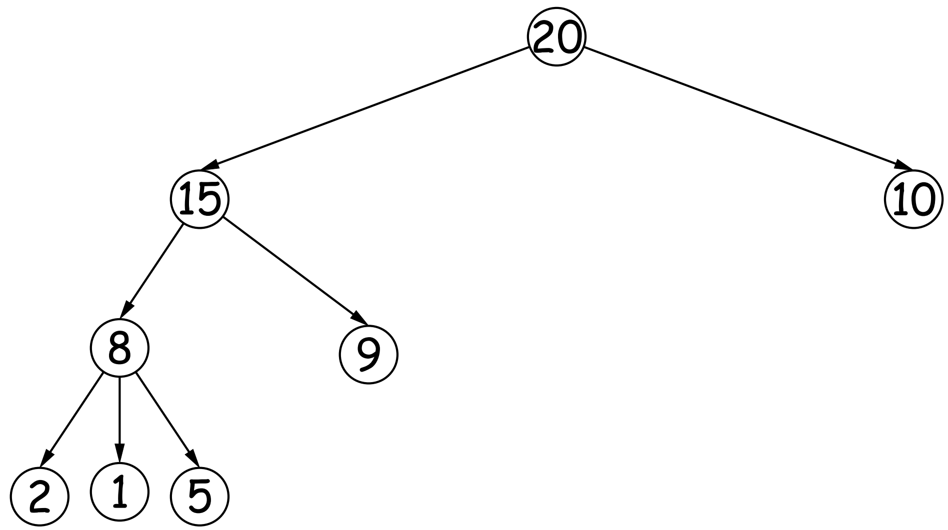
- We can have rows of rows, and rows of rows of rows, but we needn't stop at an arbitrary limit.
- Result can be thought of as a form of *tree*.
- E.g: One way to see `[[[3, 7, 8], 9], 10]`:



- The circles are called *vertices* or *nodes*, connected by *edges*.
- Top node is the *root*, bottom ones are *leaves*, non-leaves are *inner nodes*.
- Each node is itself the root of a *subtree*; those immediately below are its *children*.

Trees With Labels

- Generally, each node (not just leaves) can have additional data, known as a *label*:



- How can we represent this structure?

Tree Interface

- Evidently, trees have labels and children, suggesting an API like this:

```
def make_tree(label, branches = [])
    """A (sub)tree with given LABEL at its root, whose children
    are KIDS."""

def label(tree):
    """The label on TREE."""

def branches(tree):
    """The children of TREE (each a tree)."""

def isleaf(tree):
    """True if TREE is a leaf node."""
```

- Representation?

Tree Representation

```
def make_tree(label, kids = [])
    """A (sub)tree with given LABEL at its root, whose children
    are KIDS."""
    return [ label ] + kids

def label(tree):
    """The label on TREE."""
    return tree[0]

def branches(tree):
    """The children of TREE (each a tree)."""
    return tree[1:]

def isleaf(tree):
    """True if TREE is a leaf node."""
    return len(tree) == 1
```

Alternatives?

Tree Representation (II)

```
def make_tree(label, kids = [])
    """A (sub)tree with given LABEL at its root, whose children
    are KIDS."""
    return (label, kids)

def label(tree):
    """The label on TREE."""
    return tree[0]

def branches(tree):
    """The children of TREE (each a tree)."""
    return tree[1]

def isleaf(tree):
    """True if TREE is a leaf node."""
    return len(branches(tree)) == 0
```

Algorithms on Trees

- Trees have a recursive structure. A tree is:
  - A label and
  - Zero or more children, each a tree.
- Recursive structure implies recursive algorithm.

Counting Leaves

```
def count_leaves(tree):
    """The number of leaf nodes in TREE."""

    if ________________:

        return ________________

    else:

        return sum(________________________________)
```

Counting Leaves (II)

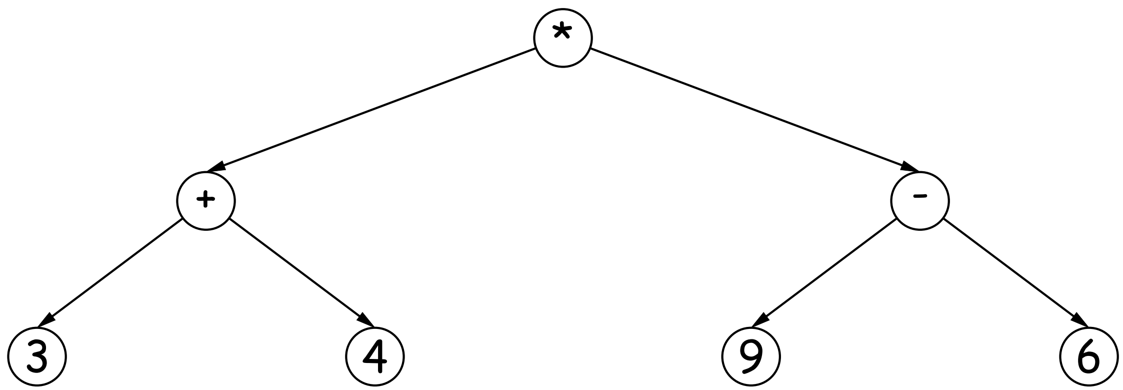
```
def count_leaves(tree):
    """The number of leaf nodes in TREE."""

    if isleaf(tree):

        return 1

    else:

        return sum(_________________________________)
```

Counting Leaves (III)

```
def count_leaves(tree):
    """The number of leaf nodes in TREE."""

    if isleaf(tree):

        return 1

    else:

        return sum(map(count_leaves, branches(tree)))

    # or

        return sum([ count_leaves(x) for x in branches(tree) ])
```

Evaluating an Expression

- Trees can represent arithmetic expressions.
- Leaf labels are numbers; other labels are operators (`+`, `-`, `*`, `/`)
- So `(3 + 4) * (9 - 6)` is



- Can we write a program to evaluate such an *expression tree* (i.e., return the value of the expression it represents)?

Evaluation

```
def value(expr):
    """Return the value of the expression represented by the
    expression tree expr
    >>> value(make_tree("*", [ make_tree("+", [make_tree(3), make_tree(4)]),
    ...                        make_tree("-", [make_tree(9), make_tree(6)])))
    36
    """
    if isleaf(expr):

        return ________________________

    elif ________________________:

        return ________________________

    ...?
```

Evaluation (II)

```
def value(expr):
    """Return the value of the expression represented by the
    expression tree expr.
    >>> value(make_tree("*", [ make_tree("+", [make_tree(3), make_tree(4)]),
    ...                        make_tree("-", [make_tree(9), make_tree(6)]))
    21
    """
    if isleaf(expr):

        return label(expr)

    elif label(expr) == '+':

        return value(branches(expr)[0]) + value(branches(expr)[1])

    ...?
```