Final Examples
Announcements
Trees
Tree-Structured Data

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

def label(tree):
    return tree[0]

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

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

A tree can contains other trees:

[5, [6, 7], 8, [[9], 10]]
(+ 5 (- 6 7) 8 (* (- 9) 10))

(S
    (NP (JJ Short) (NNS cuts))
     (VP (VBP make)
       (NP (JJ long) (NNS delays)))
    (. .))

<ul>
    <li>Midterm <b>1</b></li>
    <li>Midterm <b>2</b></li>
</ul>

Tree processing often involves recursive calls on subtrees
Tree Processing
Implement **bigs**, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    ""

    The root label is always larger than all of its ancestors
    if node.label > max(ancestors):
        ... Somehow track the largest ancestor
    if t.is_leaf():
        return ___
    else:
        return ___([___ for b in t.branches])
```

Somehow track a list of ancestors
Somehow track the largest ancestor
Somehow increment the total count

```python
if node.label > max(ancestors):
    ...
```
Solving Tree Problems

Implement \texttt{bigs}, which takes a Tree instance \( t \) containing integer labels. It returns the number of nodes in \( t \) whose labels are larger than any labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in \( t \) that are larger than all their ancestors."
    def f(a, x):
        """A node in \( t \):
        \begin{itemize}
        \item \texttt{max\_ancestor} \hspace{1cm} \texttt{node.label > max\_ancestors}
        \end{itemize}
        if \texttt{a.label > x}:
            return 1 + \sum([f(b, a.label) \text{ for } b \text{ in } a.branches])
        else:
            return \sum([f(b, x) \text{ for } b \text{ in } a.branches])
    return f(t, t.label - 1)
```

Some initial value for the largest ancestor so far...

Somehow track the largest ancestor

Root label is always larger than its ancestors

Somehow increment the total count

A node in \( t \)
Recursive Accumulation
Implement **bigs**, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0
    def f(a, x):
        nonlocal n
        if a.label > x:
            n += 1
        for b in a.branches:
            f(b, max(a.label, x))
    f(t, t.label - 1)
    return n
```
Designing Functions
How to Design Programs

From Problem Analysis to Data Definitions
Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header
State what kind of data the desired function consumes and produces. Formulate a concise answer to the question what the function computes. Define a stub that lives up to the signature.

Functional Examples
Work through examples that illustrate the function’s purpose.

Function Template
Translate the data definitions into an outline of the function.

Function Definition
Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Testing
Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

https://htdp.org/2018-01-06/Book/
Applying the Design Process
Designing a Function

Implement `smalls`, which takes a Tree instance \( t \) containing integer labels. It returns the non-leaf nodes in \( t \) whose labels are smaller than any labels of their descendant nodes.

```python
def smalls(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."
    result = []
    def process(t):
        """Find smallest label in \( t \) & maybe add \( t \) to result"
        if t.is_leaf():
            return t.label
        else:
            return min(...)
    process(t)
    return result
```

**Signature:** `Tree -> List of Trees`
Implement `smalls`, which takes a Tree instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    ""
    result = []
    def process(t):
        """Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return t.label
        else:
            smallest = min([process(b) for b in t.branches])
            if t.label < smallest:
                result.append(t.label)
            return min(smallest, t.label)
    process(t)
    return result
```

**Signature:** `Tree -> List of Trees`

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

**Signature:** `Tree -> number`

```python
if t.is_leaf():
    return t.label
else:
    smallest = min([process(b) for b in t.branches])
    if t.label < smallest:
        result.append(t.label)
    return min(smallest, t.label)
```

**Find smallest label in t & maybe add t to result**
Expression Trees
Interpreter Analysis

How many times does scheme_eval get called when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```