A tree can contain other trees: 
(5, (6, 7), 8, ([9], 10))

Tree processing often involves recursive calls on subtrees

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
```

```
def f(a, x):
    if a.is_leaf():
        return 1
    else:
        return sum(f(b, a.label) for b in a.branches) + 1
    return sum(f(b, x) for b in a.branches)
```

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

```python
if node.label > max_ancestors:
    ... # Somehow track the largest ancestor.
```

```
if node.label > max_ancestors:
    ... # Somehow track a list of ancestors.
else:
    ... # Somehow increment the total count.
```

```
A node in t
max_ancestor
```

```
if node.label > max_ancestors:
    ... # Somehow track the largest ancestor.
else:
    ... # Somehow increment the total count.
```

```
def f( , 0)
```

Recursive Accumulation
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 \texttt{bigs}(t):
    """Return the number of nodes in \( t \) that are larger than all their ancestors."

    \( n = 0 \)
    def \texttt{f}(a, x):
        \( \text{if } a.\text{label} > x \) \( n += 1 \)
        \( \text{for } b \text{ in } a.\text{branches} \) \( \text{f}(b, a.\text{label}) \)
    \( \text{return } n \)
    \texttt{f}(t, t.\text{label} - 1)
```

Somehow track the largest ancestor node. \( \text{label} > \text{max}_\text{ancestors} \)
Somehow increment the total count
Root label is always larger than its ancestor

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.

Implement \texttt{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 \texttt{smalls}(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."

    \( \text{result} = [] \)
    \( \text{def } \texttt{process}(t) \):
        \( \text{result}.\text{append}(t) \)
        \( \text{if } t.\text{is_leaf()} \) \( \text{return } t.\text{label} \)
        \( \text{smallest} = \text{min}(\text{\{process(b) for b in } t.\text{branches}\}) \)
        \( \text{if } t.\text{label} < \text{smallest} \) \( \text{return } \text{process}(\text{\{process(b) for b in } t.\text{branches}\}) \)
    \( \text{return } \text{result} \)
```

Designing a Function

Implement \texttt{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 \texttt{smalls}(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."

    \( \text{result} = [] \)
    \( \text{def } \texttt{process}(t) \):
        \( \text{result}.\text{append}(t) \)
        \( \text{if } t.\text{is_leaf()} \) \( \text{return } t.\text{label} \)
        \( \text{smallest} = \text{min}(\text{\{process(b) for b in } t.\text{branches}\}) \)
        \( \text{if } t.\text{label} < \text{smallest} \) \( \text{return } \text{process}(\text{\{process(b) for b in } t.\text{branches}\}) \)
    \( \text{return } \text{result} \)
```
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))
```