

Final Examples

Class outline:

- Trees
- Recursive accumulation
- Regular expressions
- Interpreters

Trees

Tree abstractions

Using functions:

```
def tree(label, branches=[]):  
    return [label] + list(branches)  
  
def label(t):  
    return t[0]  
  
def branches(t):  
    return t[1:]  
  
def is_leaf(t):  
    return not branches(t)
```

Using a class:

```
class Tree:  
    def __init__(self, label, branches=[]):  
        self.label = label  
        self.branches = list(branches)  
  
    def is_leaf(self):  
        return not self.branches
```

Tree-structured data

A tree is a recursive structure, where each branch may itself be a tree.

```
[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 <strong>1</strong></li>  
  <li>Midterm <strong>2</strong></li>  
</ul>
```

Tree processing often involves recursive calls on subtrees.

Solving tree problems

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```
def big(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)])])])  
    >>> big(a)  
    4  
    """
```

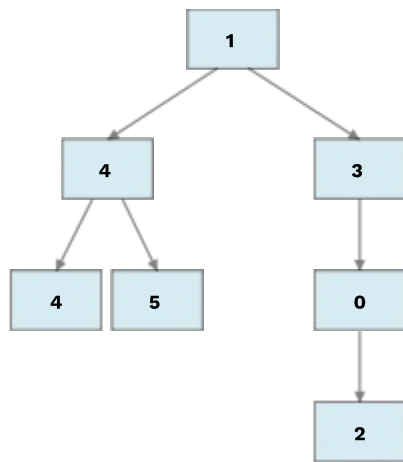
1. Understand the question and function signature.

Solving tree problems

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```
def big(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)])])])  
    >>> big(a)  
    4  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.

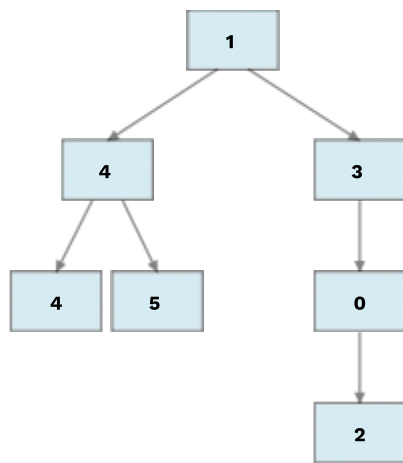


Solving tree problems

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```
def big(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)])])])  
    >>> big(a)  
    4  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.

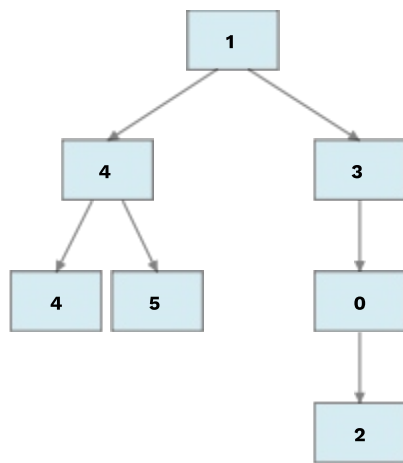


Solving tree problems

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```
def big(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)])])])  
    >>> big(a)  
    4  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.

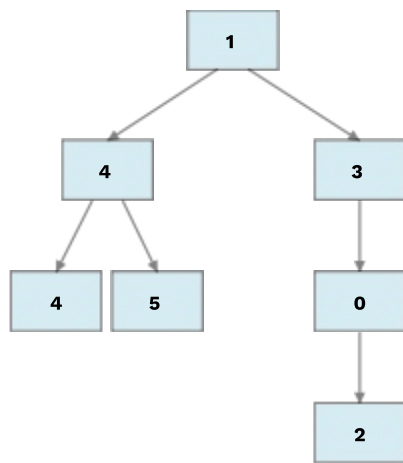


Solving tree problems

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```
def big(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)])])])  
    >>> big(a)  
    4  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.

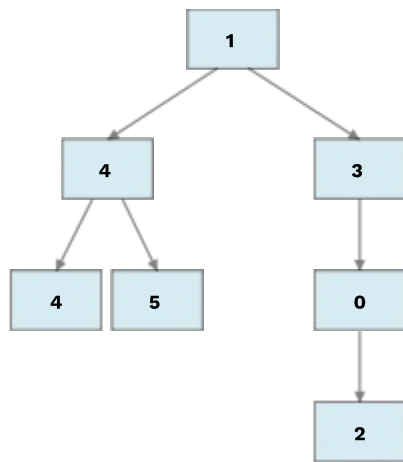


Solving tree problems

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```
def big(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)])])])  
    >>> big(a)  
    4  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.

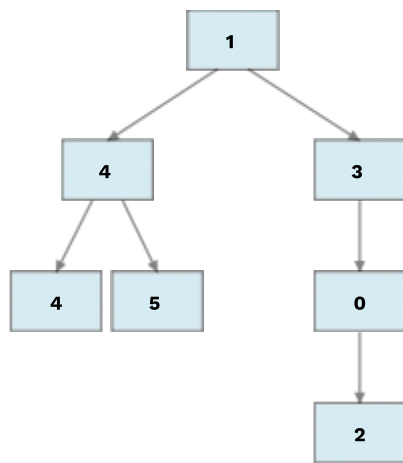


Solving tree problems

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```
def big(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)])])])  
    >>> big(a)  
    4  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.



Solving bigs #2

```
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  
    """
```

4. Consider what you expect to see in the solution.

Solving bigs #2

```
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  
    """
```

4. Consider what you expect to see in the solution.

Typical tree processing structure?

```
if t.is_leaf():  
    return ____  
else:  
    return ____([____ for b in t.branches])
```

Solving bigs #2

```
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  
    """
```

4. Consider what you expect to see in the solution.

Typical tree processing structure?

```
if t.is_leaf():  
    return ____  
else:  
    return ____([____ for b in t.branches])
```

✗ That won't work, since we need to know about ancestors.

Solving bigs #3

```
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  
    """
```

4. Consider what you expect to see in the solution.

Solving bigs #3

```
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  
    """
```

4. Consider what you expect to see in the solution.

Some code that increments the total count

```
1 + _____
```


Solving bigs #3

```
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  
    """
```

4. Consider what you expect to see in the solution.

Some code that increments the total count

```
1 + _____
```

Some way of tracking ancestor labels or max of ancestors seen so far.

```
if node.label > max(ancestors):
```

```
if node.label > max_ancestor:
```

Solving bigs #4

```
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  
    """
```

5. Check out the provided template.

```
def f(a, x):  
    if _____:  
        return 1 + _____  
    else:  
        return _____  
return _____
```

Solving bigs #4

```
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  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```
def f(a, x):  
    if _____:  
        return 1 + _____  
    else:  
        return _____  
return _____
```

Solving bigs #4

```
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  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```
def f(a, x):  
    if _____:  
        return 1 + _____ # Increment total  
    else:  
        return _____  
return _____
```

Solving bigs #4

```
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  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```
def f(a, x):  
    if _____: # Track the largest ancestor  
        return 1 + _____ # Increment total  
    else:  
        return _____  
return _____
```

Solving bigs #4

```
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  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.
7. Label any ambiguously named variables if its helpful.

```
def f(a, x):  
    if _____: # Track the largest ancestor  
        return 1 + _____ # Increment total  
    else:  
        return _____  
return _____
```

Solving bigs #4

```
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  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.
7. Label any ambiguously named variables if its helpful.

```
# a is the current subtree, x is the largest ancestor  
def f(a, x):  
    if _____: # Track the largest ancestor  
        return 1 + _____ # Increment total  
    else:  
        return _____  
return _____
```

Solving bigs #5

```
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  
    """
```

8. Finish filling in the skeleton.

```
def f(a, x):  
    if a.label > x:  
        return 1 + sum([f(b, a.label) for b in a.branches])  
    else:  
        return sum([f(b, x) for b in a.branches])  
return f(t, t.label - 1)
```


Solving bigs #6

```
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.label > x:  
            return 1 + sum([f(b, a.label) for b in a.branches])  
        else:  
            return sum([f(b, x) for b in a.branches])  
    return f(t, t.label - 1)
```

Recursive accumulation

Alternative bigs approach

Initialize some data structure to an empty/zero value, and populate it as you go.

```
def bigs(t):  
    """Return the number of nodes in t that are larger than all their ancestors."""  
    n = [0]  
    def f(a, x):  
        if _____:  
            _____  
        _____:  
            f(_____)  
    _____  
    return n[0]
```

Alternative bigs solution

Initialize some data structure to an empty/zero value, and populate it as you go.

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

Solving smalls

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.

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

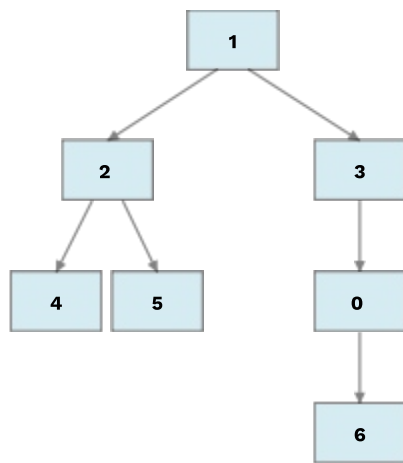
1. Understand the question and function signature.

Solving smalls

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.

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.

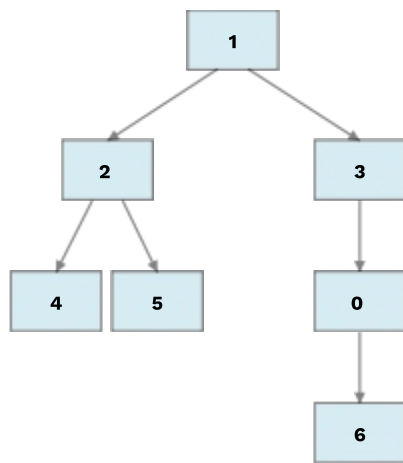


Solving smalls

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.

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.

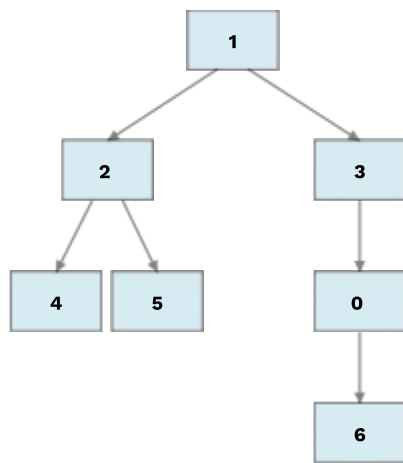


Solving smalls

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.

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.

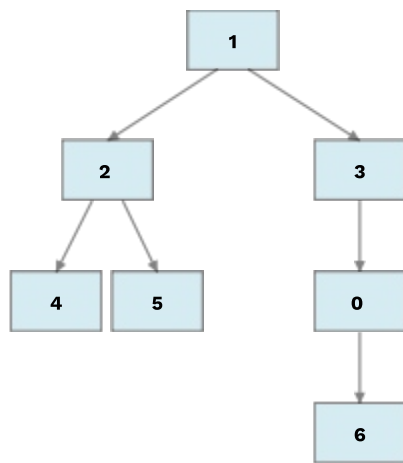


Solving smalls

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.

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.



Solving smalls #2

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

4. Consider what you expect to see in the solution.

Solving smalls #2

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

4. Consider what you expect to see in the solution.

Something which finds the smallest value in a subtree

```
min(____)
```


Solving smalls #2

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

4. Consider what you expect to see in the solution.

Something which finds the smallest value in a subtree

```
min(____)
```

Something which compares smallest to current

```
t.label < smallest
```

Solving smalls #2

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

4. Consider what you expect to see in the solution.

Something which finds the smallest value in a subtree

```
min(____)
```

Something which compares smallest to current

```
t.label < smallest
```

Something which adds a subtree to a list

```
____.append(t)
```

Solving smalls #3

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

5. Check out the provided template.

```
result = []  
def process(t):  
    if t.is_leaf():  
        return _____  
    else:  
        smallest = _____  
        if _____:  
            _____  
        return min(smallest, t.label)  
process(t)  
return result
```

Solving smalls #3

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```
result = []  
def process(t):  
    if t.is_leaf():  
        return _____  
    else:  
        smallest = _____  
        if _____:  
            _____  
        return min(smallest, t.label)  
process(t)  
return result
```

Solving smalls #3

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```
result = []  
def process(t):  
    if t.is_leaf():  
        return _____  
    else:  
        smallest = _____ # Finds smallest  
        if _____:  
            _____  
        return min(smallest, t.label)  
process(t)  
return result
```

Solving smalls #3

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```
result = []  
def process(t):  
    if t.is_leaf():  
        return _____  
    else:  
        smallest = _____ # Finds smallest  
        if _____: # Compares smallest  
            _____  
        return min(smallest, t.label)  
process(t)  
return result
```

Solving smalls #3

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```
result = [] # The result list  
def process(t):  
    if t.is_leaf():  
        return _____  
    else:  
        smallest = _____ # Finds smallest  
        if _____: # Compares smallest  
            _____ # Appends subtree to list  
        return min(smallest, t.label)  
process(t)  
return result
```

Solving smalls #3

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.
7. Label any ambiguously named variables if its helpful.

```
result = [] # The result list  
def process(t):  
    if t.is_leaf():  
        return _____  
    else:  
        smallest = _____ # Finds smallest  
        if _____: # Compares smallest  
            _____ # Appends subtree to list  
        return min(smallest, t.label)  
process(t)  
return result
```


Solving smalls #3

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.
7. Label any ambiguously named variables if its helpful.

```
result = [] # The result list  
def process(t): # t is a Tree  
    if t.is_leaf():  
        return _____  
    else:  
        smallest = _____ # Finds smallest  
        if _____: # Compares smallest  
            _____ # Appends subtree to list  
        return min(smallest, t.label)  
process(t)  
return result
```

Solving smalls #4

```
def smalls(t):  
    """Return the non-leaf nodes in t that are smaller than all their descendants.  
  
    >>> 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]  
    """
```

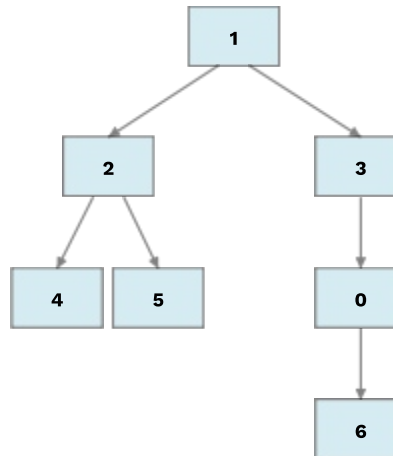
8. Finish filling in the skeleton.

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

Solving smalls #5

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

8. Check your work!



Regular expressions

Matching patterns

Which strings are matched by each regular expression?

Expressions: abc cab bac baba ababca aabcc abba

`[abc]*`

`a*b*c*`

`ab|[bc]*`

`(a[bc]+)+a?`

`(ab|ba)+`

`(ab|[bc])?`

Matching patterns

Which strings are matched by each regular expression?

| Expressions: | abc | cab | bac | baba | ababca | aabcc | abba |
|--|-----|-----|-----|------|--------|-------|------|
| <code>[abc]*</code> | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| <code>a*b*c*</code> | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| <code>ab [bc]*</code> | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| <code>(a[bc]+)+a?</code> | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ |
| <code>(ab ba)+</code> <code>(ab [bc])?</code> | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ |

Interpreters

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```


Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```


Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```


Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```

Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```