Announcements
Announcements

• Midterm 2 is on Monday 10/28 7pm–9pm
Announcements

*Midterm 2 is on Monday 10/28 7pm–9pm*

*Topics and locations: [http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html](http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html)*
Announcements

• Midterm 2 is on Monday 10/28 7pm–9pm
  • Topics and locations: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html
  • Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
Announcements

* Midterm 2 is on Monday 10/28 7pm–9pm
  * Topics and locations: [http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html](http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html)
  * Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
  * Emphasis: mutable data, object-oriented programming, recursion, and recursive data
Announcements

• Midterm 2 is on Monday 10/28 7pm–9pm
  • Topics and locations: [http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html](http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html)
  • Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
  • Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  • Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm!
Announcements

• Midterm 2 is on Monday 10/28 7pm–9pm
  • Topics and locations: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html
  • Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
  • Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  • Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm!
  • Review session on Saturday 10/26 from 1pm to 4pm in 1 Pimentel
Announcements

• Midterm 2 is on Monday 10/28 7pm–9pm
  • Topics and locations: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html
  • Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
  • Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  • Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm!
  • Review session on Saturday 10/26 from 1pm to 4pm in 1 Pimentel
  • HKN review session on Sunday 10/27 from 4pm to 7pm to 2050 VLSB
Announcements

- Midterm 2 is on Monday 10/28 7pm–9pm
  - Topics and locations: [http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html](http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html)
  - Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
  - Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  - Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm!
  - Review session on Saturday 10/26 from 1pm to 4pm in 1 Pimentel
  - HKN review session on Sunday 10/27 from 4pm to 7pm to 2050 VLSB
  - Includes content through Wednesday 10/23 (today is review & examples)
Announcements

• Midterm 2 is on Monday 10/28 7pm–9pm
  • Topics and locations: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html
  • Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
  • Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  • Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm!
  • Review session on Saturday 10/26 from 1pm to 4pm in 1 Pimentel
  • HKN review session on Sunday 10/27 from 4pm to 7pm to 2050 VLSB
  • Includes content through Wednesday 10/23 (today is review & examples)
• No lab next Monday, Tuesday, & Wednesday
Announcements

• Midterm 2 is on Monday 10/28 7pm–9pm
  • Topics and locations: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html
  • Bring 1 hand-written, 2-sided sheet of notes. Two study guides will be provided.
  • Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  • Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm!
  • Review session on Saturday 10/26 from 1pm to 4pm in 1 Pimentel
  • HKN review session on Sunday 10/27 from 4pm to 7pm to 2050 VLSB
  • Includes content through Wednesday 10/23 (today is review & examples)
• No lab next Monday, Tuesday, & Wednesday
• Homework 7 is due Tuesday 11/5 @ 11:59pm (Two weeks)
Mutable Recursive Lists
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

    >>> s = Rlist(1, Rlist(2, Rlist(3)))
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
```
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

>>> s = Rlist(1, Rlist(2, Rlist(3)))

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
```

*Note: The actual environment diagram is much more complicated.*
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
```

*Note: The actual environment diagram is much more complicated.*
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
>>> t = s.rest
```

*Note: The actual environment diagram is much more complicated.*
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
```

*Note: The actual environment diagram is much more complicated.*
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

*Note: The actual environment diagram is much more complicated.*
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of an Rlist.

The rest of a recursive list can contain the recursive list as a sub-list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

Note: The actual environment diagram is much more complicated.
Recursive Lists as Functions
Mutable Recursive Lists Using Functions

The object system is convenient, but it isn't necessary for designing data types!

(Demo)
Trees
Pruned Trees

Consider the binary Tree class below, which has no entry attribute.

class Tree(object):
    """A binary tree with no entries."""
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

a = Tree(None, Tree(Tree(), Tree(None, Tree())))
b = Tree(None, Tree())
c = Tree(None, Tree(None, Tree()))
d = Tree(Tree(), Tree())

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,c)</th>
<th>(a,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Pruned Trees

Consider the binary Tree class below, which has no entry attribute.

class Tree(object):
    """A binary tree with no entries."""
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right
a = Tree(None, Tree(Tree(), Tree(None, Tree())))
b = Tree(None, Tree())
c = Tree(None, Tree(None, Tree()))
d = Tree(Tree(), Tree())

Write a function `pruned` that takes two Tree arguments t1 and t2 and returns whether t2 is a pruned version of t1. t2 is a pruned version of t1 if all paths from the root of t2 are also valid paths from the root of t1.

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,c)</th>
<th>(a,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Pruned Tree Examples

a

b

c

d
Pruned Tree Examples

(a, b)  (a, c)  (a, d)

| pruned | True | True | False |
Recursive Idea
Recursive Idea

\[ \text{pruned}(a, c) \]
Recursive Idea

pruned(a, c)
Recursive Idea

pruned(a, c)

implies
Recursive Idea

\[ \text{pruned}(a, c) \]

implies

\[ \text{pruned}(a.\text{right}, c.\text{right}) \]
Recursive Idea

pruned(a, c)

implies

pruned(a.right, c.right)
Recursive Idea

\text{pruned}(a, c) \implies \text{pruned}(a.\text{right}, c.\text{right})
Recursive Idea

\text{pruned}(a, c) \implies \text{pruned}(a.\text{right}, c.\text{right})
Recursive Idea

pruned(a, c)

implies

pruned(a.right, c.right)

what about c.left?
Recursive Idea

pruned(a, c)

implies

pruned(a.right, c.right)

what about c.left?
Recursive Idea

pruned(a, c)

implies

pruned(a.right, c.right)

what about c.left?
Recursive Idea
Recursive Idea

\[ \text{pruned}(a, d) \]
Recursive Idea

pruned(a, d)
Recursive Idea

pruned(a, d)

would imply
Recursive Idea

\texttt{pruned(a, d)}

would imply

\texttt{pruned(a.\text{left}, d.\text{left})}
Recursive Idea

pruned(a, d)

would imply

pruned(a.left, d.left)
Recursive Idea

pruned(a, d)

would imply

pruned(a.left, d.left)
Recursive Idea

pruned(a, d)

would imply

pruned(a.left, d.left)
Recursive Idea

pruned(a, d)

would imply

pruned(a.left, d.left)
Recursive Idea

pruned(a, d)

would imply

pruned(a.left, d.left)
Recursive Implementation
Recursive Implementation

Recursive call: Both the left and right are pruned, respectively
Recursive Implementation

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None
Recursive Implementation

```
def pruned(t1, t2):
```

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

```
def pruned(t1, t2):
```
Recursive Implementation

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

```python
def pruned(t1, t2):
    if t2 is None:
        # Base case 1
    if t2 is None:
        # Base case 2
```

Diagram:

- a
- b
- c
- d

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

```python
def pruned(t1, t2):
    if t2 is None:
        # Base case 1
    if t2 is None:
        # Base case 2
```
Recursive Implementation

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

```python
def pruned(t1, t2):
    if t2 is None:
        return True
    ```
Recursive Implementation

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

```python
def pruned(t1, t2):
    if t2 is None:
        return True
    elif t1 is None:
        return True
    else:
        return pruned(t1.left, t2.left) and pruned(t1.right, t2.right)
```
Recursive Implementation

def pruned(t1, t2):
    if t2 is None:
        return True
    elif t1 is None:
        return False
    Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

def pruned(t1, t2):
    if t2 is None:
        return True
    elif t1 is None:
        return False
Recursive Implementation

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

```python
def pruned(t1, t2):
    if t2 is None:
        return True
    elif t1 is None:
        return False
    else:
        # Recursive call
        # Base cases
```
Recursive Implementation

Recursive call: Both the left and right are pruned, respectively

Base cases: one (or more) of the trees is None

```python
def pruned(t1, t2):
    if t2 is None:
        return True
    elif t1 is None:
        return False
    else:
        return pruned(t1.left, t2.left) and pruned(t1.right, t2.right)
```
Non-Local Assignment
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
```
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
return oski(abs)
```

```
Global frame
    oski
    f1: oski
    bear
    cal
    Return Value

f2: cal [parent=f1]
    berk 2
    Return Value

Return Value
```
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
    oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk + 1, berk - 1)
        bear = lambda ley: berk - ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
```
Go Bears!

def oski(bear):
    
def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
    oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)
return oski(abs)
```

```
Global frame
  oski

f1: oski
  bear
  cal
  Return Value

f2: cal [parent=f1]
  berk 2
  Return Value

f3: cal [parent=f1]
  berk 2
  Return Value

f4: λ [parent=f2]
  ley 2
  Return Value 0
```

```
tuple
  0 1
```

```
tuple
  3 1
```
Non-Local Assignment Variants
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)
```

Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return (berk+1, berk-1)
        bear = lambda ley: berk-ley
        return (berk, cal(berk))
    return cal(2)

oski(abs)