Friday, October 24
Announcements
Announcements

• Midterm 2 is on Monday 10/27 7pm–9pm
Announcements

- Midterm 2 is on Monday 10/27 7pm–9pm
  - Topics and locations: http://cs61a.org/exams/midterm2.html
Announcements

• Midterm 2 is on Monday 10/27 7pm–9pm
  ▪ Topics and locations: http://cs61a.org/exams/midterm2.html
  ▪ Bring 1 hand-written, 2-sided sheet of notes; Two study guides will be provided
Announcements

- Midterm 2 is on Monday 10/27 7pm–9pm
- Topics and locations: http://cs61a.org/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 10/27 7pm–9pm
  - Topics and locations: http://cs61a.org/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
  - Review session on Saturday 10/25 from 3pm to 6pm in 2050 VLSB
Announcements

• Midterm 2 is on Monday 10/27 7pm–9pm
  • Topics and locations: http://cs61a.org/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
  • Review session on Saturday 10/25 from 3pm to 6pm in 2050 VLSB
  • HKN review session on Sunday, 10/26 12–3pm in 155 Dwinelle
Announcements

• Midterm 2 is on Monday 10/27 7pm–9pm
  ▪ Topics and locations: http://cs61a.org/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
  ▪ Review session on Saturday 10/25 from 3pm to 6pm in 2050 VLSB
  ▪ HKN review session on Sunday, 10/26 12–3pm in 155 Dwinelle
  ▪ Includes content through Wednesday 10/22 (today is review & examples)
Announcements

• Midterm 2 is on Monday 10/27 7pm–9pm
  ▪ Topics and locations: http://cs61a.org/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
  ▪ Review session on Saturday 10/25 from 3pm to 6pm in 2050 VLSB
  ▪ HKN review session on Sunday, 10/26 12–3pm in 155 Dwinelle
  ▪ Includes content through Wednesday 10/22 (today is review & examples)
  ▪ Discussion handouts contain practice questions!
Announcements

• Midterm 2 is on Monday 10/27 7pm–9pm
  ▪ Topics and locations: http://cs61a.org/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
  ▪ Review session on Saturday 10/25 from 3pm to 6pm in 2050 VLSB
  ▪ HKN review session on Sunday, 10/26 12–3pm in 155 Dwinelle
  ▪ Includes content through Wednesday 10/22 (today is review & examples)
  ▪ Discussion handouts contain practice questions!
• No lecture next Monday, No lab next Tuesday & Wednesday, No office hours next Mon–Wed
Mutable Linked Lists
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

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

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

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

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

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

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

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

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

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

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

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(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 a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(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 a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(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 a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(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 a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(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 a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(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 a Link.

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

```python
>>> s = Link(1, Link(2, Link(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 a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(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.
Hailstone Trees
Hailstone Trees
Hailstone Trees

Pick a positive integer $n$ as the start
Hailstone Trees

Pick a positive integer n as the start

If n is even, divide it by 2
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1
Hailstone Trees

Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

1 2 4 8
Hailstone Trees

Pick a positive integer \( n \) as the start
If \( n \) is even, divide it by 2
If \( n \) is odd, multiply it by 3 and add 1
Continue this process until \( n \) is 1

1
2
4
8
16
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer \( n \) as the start

1

If \( n \) is even, divide it by 2

2

If \( n \) is odd, multiply it by 3 and add 1

4

Continue this process until \( n \) is 1

8

16

32

64
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

1
2
4
8
16
32
64
128
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

1
2
4
8
16
32
64
128
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

1
2
4
8
16
32
64
128
5
Hailstone Trees

Pick a positive integer \(n\) as the start
If \(n\) is even, divide it by 2
If \(n\) is odd, multiply it by 3 and add 1
Continue this process until \(n\) is 1
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer \( n \) as the start
If \( n \) is even, divide it by 2
If \( n \) is odd, multiply it by 3 and add 1
Continue this process until \( n \) is 1

All possible \( n \) that start a length-8 hailstone sequence
Hailstone Trees

Pick a positive integer $n$ as the start
- If $n$ is even, divide it by 2
- If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

**Question:** Write `hailstone_tree(k, n=1)`, which returns a Tree in which the paths from the leaves to the root are all possible hailstone sequences of length $k$ ending in $n$.

All possible $n$ that start a length-8 hailstone sequence

```
    1
   /|
  2/ \
 4/   8/|
32/     5/ |
64/       10/    128/      21/  20/  3
```
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

**Question:** Write `hailstone_tree(k, n=1)`, which returns a Tree in which the paths from the leaves to the root are all possible hailstone sequences of length \( k \) ending in \( n \).

All possible \( n \) that start a length-8 hailstone sequence

(Demo)
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

Question: Write $\text{hailstone\_tree}(k, n=1)$, which returns a Tree in which the paths from the leaves to the root are all possible hailstone sequences of length $k$ ending in $n$.

All possible $n$ that start a length-8 hailstone sequence

(Demo)

Question: List the leaves of a Tree instance
Binary Hailstone Trees
Hailstone Trees using BinaryTree

```
1
 | 
2
 | 
4
 | 
8
 | 
16
 | 
32  5
 | 
64  10
 | 
128 21 20 3
```
Hailstone Trees using BinaryTree

class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]
Hailstone Trees using BinaryTree

class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

@property
def left(self):
    return self.branches[0]

@property
def right(self):
    return self.branches[1]
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

@property
    def left(self):
        return self.branches[0]

@property
    def right(self):
        return self.branches[1]
Hailstone Trees using BinaryTree

class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]

(Demo)
Hailstone Trees using BinaryTree

class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]

(Demo)

**Question:** List the entries in the longest path in a binary tree for which all elements are less than k
Environment Diagrams
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!

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)
occupy(abs)

Return Value

Global frame
oski

f1: oski [parent=G]
    bear
    cal
    Return Value

f2: cal [parent=f1]
    berk 2
    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)
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!

```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)
```

Diagram showing the frame structure and call stack for the functions `oski` and `cal`.
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 cal(abs)
```

Diagram of the function calls and variable bindings:

- `func oski(bear)[parent=G]`
- `func cal(berk)[parent=f1]`
- `func lambda ley: berk-ley[parent=f2]`
- `Global frame oski`

The function `oski(abs)` is called, and its body includes a nested function `cal(berk)` which iteratively applies the function `bear` until it returns a zero value.

The output of `oski(abs)` is determined by the final value of `cal(2)`.

The diagram illustrates the flow of control and the bindings of variables throughout the function calls.
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)
```

```
Global frame
oski

f1: oski [parent=G]
    bear
    cal
    Return Value

f2: cal [parent=f1]
    berk
    Return Value

Return Value
```

---

```
func oski(bear)[parent=G]
func λ(ley) [parent=f2]
func cal(berk) [parent=f1]
```
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)

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!

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)