Monday, March 16
Announcements
Announcements

• Midterm 2 is on Thursday 3/19 7pm–9pm
Announcements

• Midterm 2 is on Thursday 3/19 7pm–9pm
  ▪ Topics and locations: http://cs61a.org/exams/midterm2.html
Announcements

• Midterm 2 is on Thursday 3/19 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 Thursday 3/19 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 Thursday 3/19 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 Tuesday 5:00pm–6:30pm in 2050 VLSB
Announcements

• Midterm 2 is on Thursday 3/19 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 Tuesday 5:00pm–6:30pm in 2050 VLSB
  ▪ Includes content through Friday 3/13 (today is review & examples)
Announcements

• Midterm 2 is on Thursday 3/19 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 Tuesday 5:00pm–6:30pm in 2050 VLSB
  ▪ Includes content through Friday 3/13 (today is review & examples)
• No lecture next Wednesday 3/18
Announcements

• Midterm 2 is on Thursday 3/19 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 Tuesday 5:00pm–6:30pm in 2050 VLSB
  ▪ Includes content through Friday 3/13 (today is review & examples)
• No lecture next Wednesday 3/18
• No discussion sections next Thursday 3/19 or Friday 3/20
Announcements

• Midterm 2 is on Thursday 3/19 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 Tuesday 5:00pm-6:30pm in 2050 VLSB
  ▪ Includes content through Friday 3/13 (today is review & examples)

• No lecture next Wednesday 3/18

• No discussion sections next Thursday 3/19 or Friday 3/20

• Lecture next Friday 3/20 is a video (but a great one)
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

>>> 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)))
>>> 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.
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)
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)
```

Return Value

Global frame

```
func oski(bear)[parent=G]

f1: oski [parent=G]
   bear
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
      ________________________
Return Value
```

Return Value

Return Value

Return Value
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)
```

Return Value
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)
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)
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)
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)
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)

oski(abs)
```

```
Global frame
oski

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

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

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

f4: λ [parent=f2]
    Return Value
```
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)
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)
oaksi(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)
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)
Objects
Land Owners

Instance attributes are found before class attributes; class attributes are inherited
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```python
>>> Worker().work()
>>> jack
>>> jack.work()
>>> john.work()
>>> john.elf.work(john)
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()  # >>> Worker().work()
john = Bourgeoisie()  # <<< jack
jack.greeting = 'Maam'
>>> jack.work()
>>> john.work()
>>> john.elf.work(john)
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
greeting = 'Sir'
def __init__(self):
    self.elf = Worker
def work(self):
    return self.greeting + ', I work'
def __repr__(self):
    return Bourgeoisie.greeting

class Bourgeoisie(Worker):
greeting = 'Peon'
def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker() >>> Worker().work() 
john = Bourgeoisie() >>> jack
jack.greeting = 'Maam' >>> jack.work() 
john = Bourgeoisie() >>> john.work() 
john.elf = Worker() >>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()  <class Worker>
    greeting: 'Sir'

>>> jack  
<class Bourgeoisie>
    greeting: 'Peon'

>>> jack.work()  jack <Worker>
    elf:

>>> john.work()  

>>> john.elf.work(john)
Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

>>> Worker().work()  
``<class Worker>```  
greeting: 'Sir'  

>>> jack  
``<class Bourgeoisie>```  
greeting: 'Peon'  

>>> jack.work()  
jack <Worker>  
elf:  

>>> john.work()  
``<class Bourgeoisie>```  

elf:  

>>> john.elf.work(john)  
``john <Bourgeoisie>```  
elf:  

Land Owners

Instance attributes are found before class attributes; class attributes are inherited
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()  <class Worker>
    greeting: 'Sir'

>>> jack
    <class Bourgeoisie>
    greeting: 'Peon'

>>> jack.work()

>>> john.work()
    <class Bourgeoisie>
    elf: john
    greeting: 'Maam'

>>> john.elf.work(john)
    <class Bourgeoisie>
### Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie'.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'
jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```python
>>> Worker().work()  
<class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()  

```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

>>> Worker().work()
'Sir, I work'

>>> jack
<class Worker>
greeting: 'Sir'

>>> jack.work()
<class Bourgeoisie>
greeting: 'Peon'

>>> john.work()
<class Bourgeoisie>
greeting: 'Maam'

>>> john.elf.work(john)
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
def __init__(self):
    self.elf = Worker
def work(self):
    return self.greeting + ', I work'
def __repr__(self):
    return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
<class Worker>
greeting: 'Sir'

>>> jack.work()
<class Bourgeoisie>
greeting: 'Peon'

>>> john.work()
<class Bourgeoisie>
egreeting: 'Maam'

>>> john.elf.work(john)

jack <Worker>
elf: 

greeting: 'Maam'

john <Bourgeoisie>
elf: 

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

jack
Peon

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

jack <Worker>

elf:

greeting: 'Maam'

john <Bourgeoisie>

elf:
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)
```

```text
<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

greeting: 'Maam'

jack <Worker>
elf: 
greeting: 'Maam'

john <Bourgeoisie>
elf: 
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.' + self.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```python
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()

>>> john.elf.work(john)
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
greeting = 'Sir'
def __init__(self):
    self.elf = Worker
def work(self):
    return self.greeting + ', I work'
def __repr__(self):
    return Bourgeoisie.greeting

class Bourgeoisie(Worker):
greeting = 'Peon'
def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()

>>> john.elf.work(john)

jake <Worker>
greeting: 'Peon'

john <Bourgeoisie>
greeting: 'Maam'

elf:  

d['elf':  

jack <Worker>
greeting: 'Sir'

elf:  

d['elf':  

john <Bourgeoisie>
greeting: 'Maam'

elf:  

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work
'I gather wealth'

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

ejack <Worker>
egreeting: 'Peon'

<class Bourgeoisie>
greeting: 'Peon'
jack <Worker>
elf: 

greeting: 'Maam'

john <Bourgeoisie>
elf: 

john <Bourgeoisie>
**Land Owners**

Instance attributes are found before class attributes; class attributes are inherited.

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work
'I gather wealth'

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

jack <Worker>
elf:

greeting: 'Maam'

john <Bourgeoisie>
elf:
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work'

>>> john.elf.work(john)
'Peon, I work'
Binary Trees
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

A:   ● ●
B:   ● ● ● ●
C:   ● ● ● ● ●
D:   ● ● ●
E:   ●
...

...
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

Problem: Implement `morse` so that `decode` works correctly.

```
  A:  ●●
  B:  ● ●● ●
  C:  ● ● ●●
  D:  ● ●●
  E:  ●
```

...
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```
abe = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter using a Morse code tree."""
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter using a morse code tree."
    >>> t = morse(abcde)
```

```python
A: ●●
B: ●●●●
C: ●●●●●●
D: ●●●●
E: ●
...```

```python
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.
```

```python
def decode(signals, tree):
    """Decode signals into a letter using a morse code tree."

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ["", "", "", "", "", ""]]
    ["d", "e", "c", "a", "d", "e"]
```

A: ● ●
B: ● ● ● ●
C: ● ● ● ● ●
D: ● ● ● ●
E: ●
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

Problem: Implement `morse` so that `decode` works correctly.

```python
def decode(signals, tree):
    # Decode signals into a letter using a morse code tree.
    #
    # >>> t = morse(abcde)
    # >>> [decode(s, t) for s in ['--..', '.', '-.--', '.-', '-..']]  #
    # ['d', 'e', 'c', 'a', 'd', 'e']
    
    for signal in signals:
        #
```

```python
def morse(abcde):
    # Morse code dictionary
    morse_code = {  
        'a': '.-',  
        'b': '-...',  
        'c': '-.-.',  
        'd': '-..',  
        'e': '.',  
    }
    
    # Implement morse
    return morse_code

# Example usage
morse_code = morse(abcde)
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

Problem: Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    # Decode signals into a letter using a morse code tree.
    for signal in signals:
        if signal == '.':
            tree = tree.left
```

A:  [●●]
B:   [●●●●]
C:    [●●●●●]
D:     [●●●]
E:      [●]
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.', 'b': '-...', 'c': '-.-', 'd': '-..', 'e': ' '}

def decode(signals, tree):
    """Decode signals into a letter using a morse code tree.""
    for signal in signals:
        if signal == '.':
            tree = tree.left
        elif signal == '-':
            tree = tree.right
```

```python
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter using a morse code tree."

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']

    for signal in signals:
        if signal == '.':
            tree = tree.left
        elif signal == '-':
            tree = tree.right
    return tree.entry
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement \texttt{morse} so that \texttt{decode} works correctly

\begin{verbatim}
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def \texttt{decode}(signals, tree):
    """Decode signals into a letter using a morse code tree."

    >>> t = \texttt{morse}(abcde)
    >>> [\texttt{decode}(s, t) for s in ['-', '.', '-.-', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']

    for signal in signals:
        if signal == '.':
            tree = tree.left
        elif signal == '-':
            tree = tree.right
    return tree.entry
\end{verbatim}
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem**: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.}

def decode(signals, tree):
    """Decode signals into a letter using a morse code tree."

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in [-.., , , , , , , -..]]
    ['d', 'e', 'c', 'a', 'd', 'e']

    for signal in signals:
        if signal == '.':
            tree = tree.left
        elif signal == '-':
            tree = tree.right
    return tree.entry
```

```python
def morse(code):
    ....
(Demo)```