61A Lecture 18

Friday, March 6
Announcements
Announcements

*Project 3 due Thursday 3/12 @ 11:59pm (get started now!)*
Announcements

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
Announcements

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  
  ▪ Bonus point for early submission by Wednesday 3/11

• Homework 6 due Monday 3/16 @ 11:59pm (not yet released)
Announcements

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  ▪ Bonus point for early submission by Wednesday 3/11

• Homework 6 due Monday 3/16 @ 11:59pm (not yet released)

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

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  ▪ Bonus point for early submission by Wednesday 3/11
• Homework 6 due Monday 3/16 @ 11:59pm (not yet released)
• Midterm 2 is on Thursday 3/19 7pm–9pm
  ▪ Emphasis: mutable data, object-oriented programming, recursion, and recursive data
Announcements

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  ▪ Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  ▪ Bonus point for early submission by Wednesday 3/11
• Homework 6 due Monday 3/16 @ 11:59pm (not yet released)
• Midterm 2 is on Thursday 3/19 7pm–9pm
  ▪ Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  ▪ Fill out conflict form if you cannot attend due to a course conflict
Hog Contest Results
Hog Contest Results

Excellent participation!

51 qualified submissions

Lots of excellent ideas
Hog Contest Results

Excellent participation!
51 qualified submissions
Lots of excellent ideas
Hog Contest Results

Excellent participation!
51 qualified submissions
Lots of excellent ideas

(Results)
Type Coercion
Review: Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary.

Extensible: Any new numeric type can "install" itself into the existing system by adding new entries to the cross-type function dictionaries.
Review: Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary.

Extensible: Any new numeric type can "install" itself into the existing system by adding new entries to the cross-type function dictionaries

<table>
<thead>
<tr>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coercion
Coercion

Idea: Some types can be converted into other types
Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system
Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```python
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```
Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```python
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?
Coercion

Idea: Some types can be converted into other types
Takes advantage of structure in the type system

```python
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?

Question: Can any two numeric types be coerced into a common type?
Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```python
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?

Question: Can any two numeric types be coerced into a common type?

Question: Is coercion exact?
Applying Operators with Coercion
Applying Operators with Coercion

```python
class Number:
```
Applying Operators with Coercion

class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)
Applying Operators with Coercion

class Number:
    def add(self, other):
        x, y = self.coerce(other)
        return x.add(y)

Always defer to add method
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)
    def coerce(self, other):

Always defer to add method
Applying Operators with Coercion

```python
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)
    
def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
```

Always defer to add method
Applying Operators with Coercion

```python
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
```

Always defer to `add` method

Same interface: no change required
Applying Operators with Coercion

class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return x.add(y)
Applying Operators with Coercion

class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            Always defer to add method
            return rational_to_complex
        elif (other.type_tag, self.type_tag) in self.coercions:
            Same interface: no change required
        else:
            raise TypeError

coercions = {('rat', 'com'): rational_to_complex}
Applying Operators with Coercion

class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)

coercions = {('rat', 'com'): rational_to_complex}
Applying Operators with Coercion

class Number:
    def add(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)

    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

coercions = {('rat', 'com'): rational_to_complex}
Applying Operators with Coercion

class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)
        elif (other.type_tag, self.type_tag) in self.coercions:
            return (self, other.coerce_to(self.type_tag))

    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

coercions = {('rat', 'com'): rational_to_complex}

Always defer to add method

Same interface: no change required

coerce_to()
Applying Operators with Coercion

```python
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)
        elif (other.type_tag, self.type_tag) in self.coercions:
            return (self, other.coerce_to(self.type_tag))

    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

coercions = {('rat', 'com'): rational_to_complex}
```

(Demo)
Coercion Analysis
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

<table>
<thead>
<tr>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

<table>
<thead>
<tr>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Coerce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
</tr>
</tbody>
</table>
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme
Linked Lists
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list
A linked list is either empty or a first value and the rest of the linked list

3, 4, 5
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

\[ 3, 4, 5 \]

**Link instance**

<table>
<thead>
<tr>
<th>first</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rest</td>
<td></td>
</tr>
</tbody>
</table>

The diagram illustrates a link instance with the first value being 3 and the rest being empty.
A linked list is either empty or a first value and the rest of the linked list.

3, 4, 5
A linked list is either empty or a first value and the rest of the linked list.

```
3, 4, 5
```

**Linked List Structure**

<table>
<thead>
<tr>
<th>Link instance</th>
<th>first:</th>
<th>rest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>first:</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>rest:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link instance</th>
<th>first:</th>
<th>rest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>first:</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>rest:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link instance</th>
<th>first:</th>
<th>rest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>first:</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>rest:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list.

3, 4, 5
**Linked List Structure**

A linked list is either empty or a first value and the rest of the linked list.

A linked list is a pair.

```
3, 4, 5
```

<table>
<thead>
<tr>
<th>Link instance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first</strong>: 3</td>
</tr>
<tr>
<td><strong>rest</strong>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link instance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first</strong>: 4</td>
</tr>
<tr>
<td><strong>rest</strong>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link instance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first</strong>: 5</td>
</tr>
<tr>
<td><strong>rest</strong>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link.empty</td>
</tr>
</tbody>
</table>
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

A linked list is a pair

The first (zeroth) element is an attribute value

3, 4, 5
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list.

A linked list is a pair

The first (zeroth) element is an attribute value

The rest of the elements are stored in a linked list
A linked list is either empty or a first value and the rest of the linked list.

A linked list is a pair:
- The first (zeroth) element is an attribute value.
- The rest of the elements are stored in a linked list.

A class attribute represents an empty linked list.

3, 4, 5
A linked list is either empty or a first value and the rest of the linked list

\[
\text{Link}(3, \text{Link}(4, \text{Link}(5, \text{Link}.\text{empty})))
\]
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

3, 4, 5

Link instance

| first: | 3 |
| rest:  |

Link instance

| first: | 4 |
| rest:  |

Link instance

| first: | 5 |
| rest:  |

Link.empty

Link(3, Link(4, Link(5, Link.empty)))
A linked list is either empty or a first value and the rest of the linked list.

```
3, 4, 5
```

![Diagram of linked list structure with values 3, 4, and 5]
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list.

\[ \text{Link instance} \]

<table>
<thead>
<tr>
<th>first</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Link(3, Link(4, Link(5, Link.empty)))} \]
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

\[ \text{Link(3, Link(4, Link(5, Link.empty)))} \]
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list.

\[3, 4, 5\]

\[
\text{Link instance} \\
\begin{array}{|c|}
\hline
\text{first:} & 3 \\
\hline
\text{rest:} & \text{Link instance} \\
\hline
\end{array}
\]

\[
\text{Link instance} \\
\begin{array}{|c|}
\hline
\text{first:} & 4 \\
\hline
\text{rest:} & \text{Link instance} \\
\hline
\end{array}
\]

\[
\text{Link instance} \\
\begin{array}{|c|}
\hline
\text{first:} & 5 \\
\hline
\text{rest:} & \text{Link instance} \\
\hline
\end{array}
\]

\[
\text{Link(3, Link(4, Link(5, Link.empty)))}
\]
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

\[ \text{3, 4, 5} \]

\[
\text{Link instance} \\
\text{first: 3} \\
\text{rest: \_} \\
\]

\[
\text{Link instance} \\
\text{first: 4} \\
\text{rest: \_} \\
\]

\[
\text{Link instance} \\
\text{first: 5} \\
\text{rest: \_} \\
\]

\[ \text{Link(3, Link(4, Link(5)))} \]
Linked List Class

\[ \text{Link}(3, \text{Link}(4, \text{Link}(5))) \]
Linked List Class

Linked list class: attributes are passed to `__init__`

```
Link(3, Link(4, Link(5)))
```
Linked List Class

Linked list class: attributes are passed to `__init__`

```python
class Link:
    Link(3, Link(4, Link(5)))
```
Linked List Class

Linked list class: attributes are passed to __init__

```python
class Link:

    def __init__(self, first, rest=empty):
        Link(3, Link(4, Link(5)))
```
Linked List Class

Linked list class: attributes are passed to __init__

class Link:

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)

    Link(3, Link(4, Link(5)))
Linked List Class

Linked list class: attributes are passed to __init__

```python
class Link:

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

```
Link(3, Link(4, Link(5, ))))
```
Linked List Class

Linked list class: attributes are passed to `__init__`

class Link:

def __init__(self, first, rest=empty):
    assert rest is Link.empty or isinstance(rest, Link)
    self.first = first
    self.rest = rest

Link(3, Link(4, Link(5)))
Linked List Class

Linked list class: attributes are passed to `__init__`

```python
class Link:
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Returns whether `rest` is a `Link`.

`help(isinstance)`: Return whether an object is an instance of a class or of a subclass thereof.

```
Link(3, Link(4, Link(5)))
```
Linked List Class

Linked list class: attributes are passed to \_\_init\_

```python
class Link:
    empty = ()

    def \_\_init\_(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.

Link(3, Link(4, Link(5)))
```
Linked List Class

Linked list class: attributes are passed to __init__

```python
class Link:
    empty = ()  # Some zero-length sequence

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

# Example usage:
Link(3, Link(4, Link(5)))
```

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.
Linked List Class

Linked list class: attributes are passed to \_\_init\_

```python
class Link:
    empty = ()  # Some zero-length sequence

def \_\_init\_\_(self, first, rest=empty):
    assert rest is Link.empty or isinstance(rest, Link)
    self.first = first
    self.rest = rest
```

Demo:
```
Link(3, Link(4, Link(5)))
```

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.
Sequence Operations
More special method names:

`__getitem__` Element selection []

`__len__` Built-in len function
Linked List Class

Linked lists are sequences

More special method names:

__getitem__   Element selection []
__len__       Built-in len function
Linked List Class

Linked lists are sequences

class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

More special method names:

     __getitem__  Element selection []
     __len__     Built-in len function
Linked List Class

Linked lists are sequences

class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest
    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

More special method names:
    __getitem__ Element selection []
    __len__   Built-in len function
Linked List Class

Linked lists are sequences

class Link:
    empty = ()

def __init__(self, first, rest=empty):
    assert ...
    self.first = first
    self.rest = rest

def __getitem__(self, i):
    if i == 0:
        return self.first
    else:
        return self.rest[i-1]

More special method names:

__getitem__  Element selection []
__len__      Built-in len function

This element selection syntax
Linked List Class

Linked lists are sequences

```python
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert ...  # This is an assertion (assertion)
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
```

More special method names:

- `__getitem__`  Element selection `[]`
- `__len__`      Built-in `len` function
Linked List Class

Linked lists are sequences

class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

def __getitem__(self, i):
    if i == 0:
        return self.first
    else:
        return self.rest[i-1]

def __len__(self):
    return 1 + len(self.rest)
Linked List Class

Linked lists are sequences

class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest
    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)

More special method names:
    __getitem__    Element selection []
    __len__        Built-in len function

Calls this method

This element selection syntax

Recursive call to __len__
Linked List Class

Linked lists are sequences

class Link:
    empty = ()

def __init__(self, first, rest=empty):
    assert ...
    self.first = first
    self.rest = rest

def __getitem__(self, i):
    if i == 0:
        return self.first
    else:
        return self.rest[i-1]

def __len__(self):
    return 1 + len(self.rest)

More special method names:
__getitem__  Element selection []
__len__      Built-in len function

Calls this method

This element selection syntax

Methods can be recursive too!

(Demo)
Linked List Processing