61A Lecture 18

Friday, March 6
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

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

<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

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

```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)

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

Always defer to add method

Same interface: no change required

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

```
Link(3, Link(4, Link(5, Link.empty)))
```

The first (zeroth) element is an attribute value

A class attribute represents an empty linked list

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.

\[
\text{Link instance} \quad \text{Link instance} \quad \text{Link instance} \\
\text{first:} \quad 3 \quad \text{first:} \quad 4 \quad \text{first:} \quad 5 \\
\text{rest:} \quad \text{rest:} \quad \text{rest:} \\
\]

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

Linked list class: attributes are passed to \texttt{\_\_init\_}


class Link:

\begin{itemize}
\item \texttt{empty} \texttt{=} ()
\end{itemize}

\begin{itemize}
\item \texttt{def \_\_init\_}(self, first, rest=empty):
\item \texttt{assert rest is Link.empty or} \texttt{isinstance(rest, Link)}
\item \texttt{self.first} \texttt{=} first
\item \texttt{self.rest} \texttt{=} rest
\end{itemize}

\begin{itemize}
\item \texttt{help(isinstance)}: Return whether an object is an instance of a class or of a subclass thereof.
\end{itemize}

\texttt{Link(3, Link(4, Link(5, Link())))}

(Demo)
Sequence Operations
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

(Demo)