Announcements

- Ants project due Monday
- HW8 due next Wednesday at 7pm
- Midterm 2 next Thursday at 7pm
  - Review session Sat. 3/16 at 2pm in 2050 VLSB
  - Office hours Sun. 3/17 12-4pm in 310 Soda
  - HKN review session Sun. 3/17 at 4pm in 145 Dwinelle
  - See course website for more information

The Independence of Data Types

Data abstraction and class definitions keep types separate

Some operations need to cross type boundaries

How do we add a complex number and a rational number together?

<table>
<thead>
<tr>
<th>Rational numbers as numerators &amp; denominators</th>
<th>Complex numbers as two-dimensional vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_rational mul_rational</td>
<td>add_complex mul_complex</td>
</tr>
</tbody>
</table>

There are many different techniques for doing this!

Type Dispatching

Define a different function for each possible combination of types for which an operation (e.g., addition) is valid

```python
def iscomplex(z):
    return type(z) in (ComplexRI, ComplexMA)
def isrational(z):
    return type(z) is Rational
def add_complex_and_rational(z, r):
    return ComplexRI(z.real + r.numerator / r.denominator, z.imag)
def add_by_type_dispatching(z1, z2):
    """Add z1 and z2, which may be complex or rational."
    if iscomplex(z1) and iscomplex(z2):
        return add_complex(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        add_rational(z1, z2)
```

Tag-Based Type Dispatching

Idea: Use dictionaries to dispatch on type (like we did for message passing)

```python
def type_tag(x):
    return type_tags[type(x)]
type_tags = {ComplexRI: 'com', ComplexMA: 'com', Rational: 'rat'}
def add(x1, x2):
    types = (type_tag(x1), type_tag(x2))
    return add_implementations[types](x1, x2)
def add_imp(t1, t2):
    return lambda x, y: add_complex_and_rational(x, y)
def add_complex_and_rational(x1, x2):
    return ComplexRI(x1.real + x2.numerator / x2.denominator, x1.imag)
def add_rational(x1, x2):
    return Rational(x1.numerator + x2.numerator, x1.denominator)
def add_complex(x1, x2):
    return ComplexMA(x1.real + x2.real, x1.imag + x2.imag)
def add_implementations(types):
    return {types: add_imp}
```

Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary, but use abstract data types

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

```python
def add(x1, x2):
    types = (type_tag(x1), type_tag(x2))
    return add_implementations[types](x1, x2)
def add_complex(x1, x2):
    return ComplexMA(x1.real + x2.real, x1.imag + x2.imag)
def add_rational(x1, x2):
    return Rational(x1.numerator + x2.numerator, x1.denominator)
```

Question: How many cross-type implementations are required to support $m$ types and $n$ operations?

```
integer, rational, real, complex
m \cdot (m - 1) \cdot n
```

```
4 \cdot (4 - 1) \cdot 4 = 48
```
Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary, but use abstract data types

Extensible: Any new numeric type can "install" itself into the existing system by adding new entries to various 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>

Message Passing

Data-Directed Programming

There's nothing addition-specific about add

Idea: One dispatch function for (operator, types) pairs

```python
def apply(operator_name, x, y):
    tags = (type_tag(x), type_tag(y))
    key = (operator_name, tags)
    return apply_implementations[key](x, y)
```

```python
apply_implementations = {
    ('add', ('com', 'com')): add_complex,
    ('add', ('rat', 'rat')): add_rational,
    ('add', ('com', 'rat')): add_complex_and_rational,
    ('add', ('rat', 'com')): add_rational_and_complex,
    ('mul', ('com', 'com')): mul_complex,
    ('mul', ('rat', 'rat')): mul_rational,
    ('mul', ('com', 'rat')): mul_complex_and_rational,
    ('mul', ('rat', 'com')): mul_rational_and_complex,
}
```

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```python
def rational_to_complex(x):
    return ComplexRI(x.numerator / x.denominator, 0)
```

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

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

Question: Have we been repeating ourselves with data-directed programming?

Applying Operators with Coercion

1. Attempt to coerce arguments into values of the same type

2. Apply type-specific (not cross-type) operations

```python
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
        if (tx, ty) in coercions:
            tx, x = ty, coercions[(tx, ty)](x)
        elif (ty, tx) in coercions:
            ty, y = tx, coercions[(ty, tx)](y)
        else:
            return 'No coercion possible.'
    assert tx == ty
    key = (operator_name, tx)
    return coerce_implementations[key](x, y)
```

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:

```
1 —— 2 —— 3 —— 4 None
```

Recursive lists are recursive: the rest of the list is a list.

Nested pairs (old): (1, (2, (3, (4, None))))

Rlist class (new): Rlist(1, Rlist(2, Rlist(3, Rlist(4))))
Recursive List Class

Methods can be recursive as well!

class Rlist(object):
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()

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

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

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

There's the base case!

Yes, this call is recursive

Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

>>> s = Rlist(Rlist(1, Rlist(2, Rlist(3))))
>>> s.rest
Rlist(2, Rlist(3))
>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))

def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))

Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
    if fn(s.first):
        return Rlist(s.first, rest)
    return rest