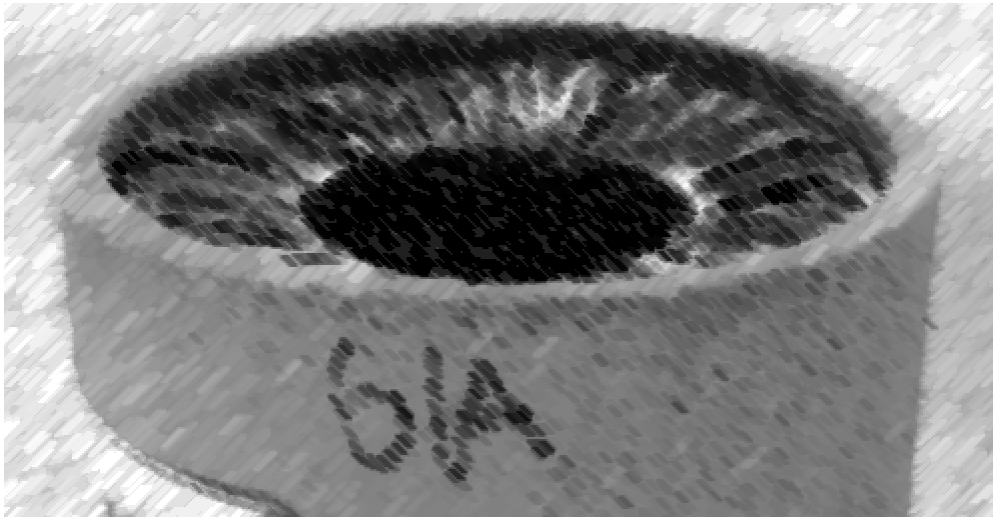


CS61A Lecture 23

Amir Kamil

UC Berkeley

March 15, 2013

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?

`add_rational  mul_rational`

Rational numbers as numerators & denominators

`add_complex  mul_complex`

Complex numbers as two-dimensional vectors

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

```
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)
```

Converted to a real number (float)

Tag-Based Type Dispatching

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

```
def type_tag(x):
    return type_tags[type(x)]

type_tags = {ComplexRI:  'com',
             ComplexMA:  'com',
             Rational:   'rat'}
```

Declares that **ComplexRI** and **ComplexMA** should be treated uniformly

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)

add_implementations = {}
add_implementations[('com', 'com')] = add_complex
add_implementations[('rat', 'rat')] = add_rational
add_implementations[('com', 'rat')] = add_complex_and_rational
add_implementations[('rat', 'com')] = add_rational_and_complex
```

```
lambda r, z: add_complex_and_rational(z, r)
```

Type Dispatching Analysis

Type Dispatching Analysis

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

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

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

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)
```

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

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)
```

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

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

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)
```

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

$$m \cdot (m - 1) \cdot n$$

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

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)
```

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

$$m \cdot (m-1) \cdot n$$

$$4 \cdot (4-1) \cdot 4 = 48$$

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

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)
```

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

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)
```

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

integer, rational, real, complex

$$m \cdot (m-1) \cdot n$$

add, subtract, multiply, divide

$$4 \cdot (4-1) \cdot 4 = 48$$

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

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

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

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

Arg 1	**Arg 2**	**Add**	**Multiply**
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

Type Dispatching

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

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

Type Dispatching

Message Passing

Data-Directed Programming

Data-Directed Programming

There's nothing addition-specific about **add**

Data-Directed Programming

There's nothing addition-specific about **add**

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

Data-Directed Programming

There's nothing addition-specific about **add**

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

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

Data-Directed Programming

There's nothing addition-specific about **add**

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

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

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

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

```
def rational_to_complex(x):
```

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

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

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

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

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

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

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

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

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

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

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

Applying Operators with Coercion

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

Applying Operators with Coercion

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

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

Applying Operators with Coercion

1. Attempt to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```
def coerce_apply(operator_name, x, y):
```

Applying Operators with Coercion

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

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

```
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
```

Applying Operators with Coercion

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

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

```
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
```

Applying Operators with Coercion

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

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

```
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
        if (tx, ty) in coercions:
```

Applying Operators with Coercion

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

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

```
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)
```

Applying Operators with Coercion

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

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

```
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:
```

Applying Operators with Coercion

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

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

```
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)
```

Applying Operators with Coercion

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

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

```
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:
```

Applying Operators with Coercion

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

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

```
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.'
```

Applying Operators with Coercion

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

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

```
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
```

Applying Operators with Coercion

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

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

```
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)
```

Applying Operators with Coercion

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

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

```
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)
```

Coercion Analysis

Coercion Analysis

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

Coercion Analysis

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

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, but use abstract data types

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, but use abstract data types

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

Coercion Analysis

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

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

From	To	Coerce
Complex	Rational	
Rational	Complex	

Coercion Analysis

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

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

From	To	Coerce
Complex	Rational	
Rational	Complex	

Coercion Analysis

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

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

From	To	Coerce
Complex	Rational	
Rational	Complex	

Type	Add	Multiply
Complex		
Rational		

Coercion Analysis

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

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	**Arg 2**	**Add**	**Multiply**
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

From	**To**	**Coerce**
Complex	Rational	
Rational	Complex	

Type	**Add**	**Multiply**
Complex		
Rational		

Closure Property of Data

Closure Property of Data

A tuple can contain another tuple as an element.

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

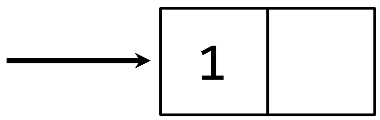
Recursive list representation of the sequence 1, 2, 3, 4:

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:

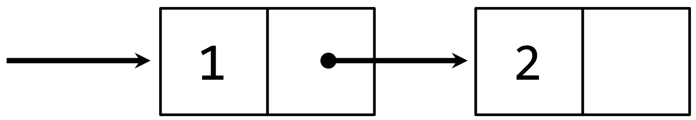


Closure Property of Data

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:

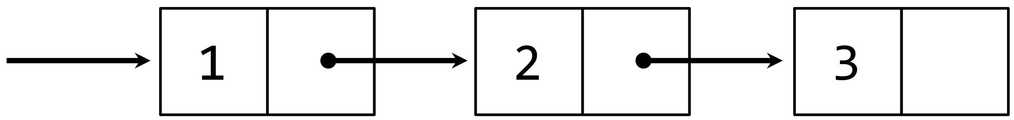


Closure Property of Data

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:

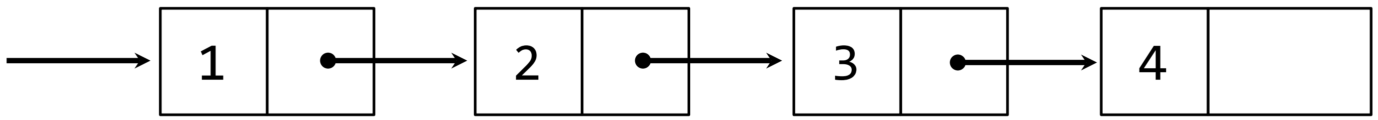


Closure Property of Data

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:

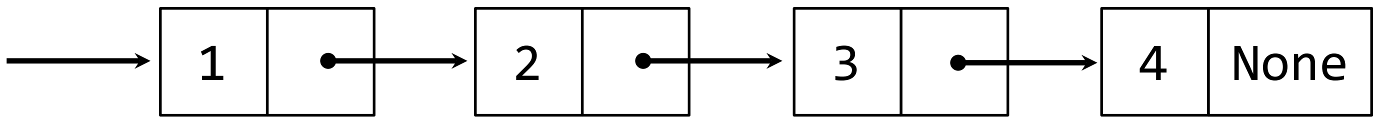


Closure Property of Data

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:

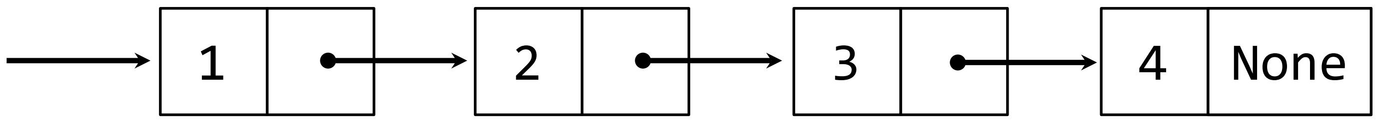


Closure Property of Data

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:



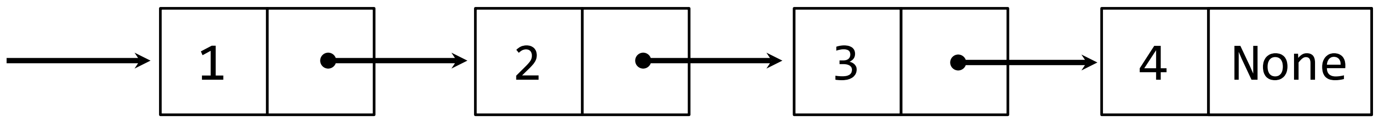
Recursive lists are recursive: the rest of the list is a list.

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:



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

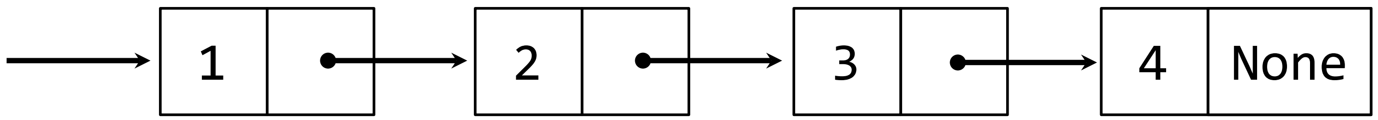
Nested pairs (old):

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:



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

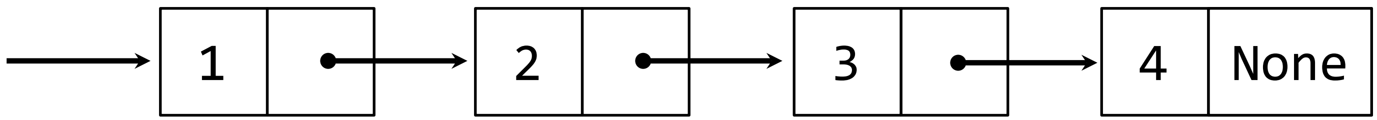
Nested pairs (old): `(1, (2, (3, (4, None))))`

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:



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

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

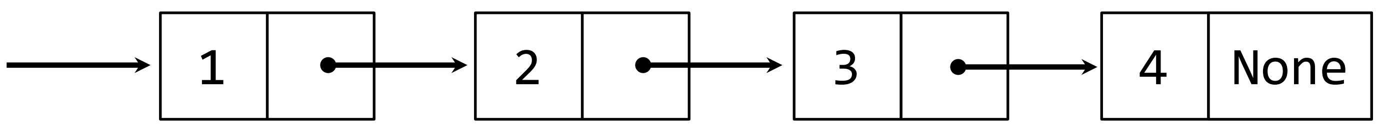
Rlist class (new):

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:



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

Recursive List Class

```
class Rlist(object):
```

Recursive List Class

```
class Rlist(object):
    class EmptyList(object):
```

Recursive List Class

```
class Rlist(object):
    class EmptyList(object):
        def __len__(self):
```

Recursive List Class

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

Recursive List Class

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

Recursive List Class

```
class Rlist(object):
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()
    def __init__(self, first, rest=empty):
```

Recursive List Class

```
class Rlist(object):
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()
    def __init__(self, first, rest=empty):
        self.first = first
```

Recursive List Class

```
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
```

Recursive List Class

```
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):
```

Recursive List Class

```
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)
```

Recursive List Class

```
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):
```

Recursive List Class

```
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:
```

Recursive List Class

```
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
```

Recursive List Class

```
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]
```

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]
```

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]
```

Yes, this call is recursive

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 Operations on Rlists

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

Recursive Operations on Rlists

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

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))
```

Recursive Operations on Rlists

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

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
```

Recursive Operations on Rlists

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

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))
```

Recursive Operations on Rlists

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

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
```

Recursive Operations on Rlists

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

```
>>> s = 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)))))
```

Recursive Operations on Rlists

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

```
>>> s = 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):
```

Recursive Operations on Rlists

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

```
>>> s = 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:
```

Recursive Operations on Rlists

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

```
>>> s = 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
```

Recursive Operations on Rlists

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

```
>>> s = 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

Map and Filter on Rlists

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

Map and Filter on Rlists

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

```
def map_rlist(s, fn):
```

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:
```

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
```

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))
```

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):
```

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:
```

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
```

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)
```

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):
```

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)
```

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
```