61A Lecture 14

Friday, September 28

Testing for Identity

Demo

Implementing Dice

Implementing Dice

Random numbers are useful for experimentation

Implementing Dice

Random numbers are useful for experimentation

They also appear in lots of algorithms, e.g.,

Implementing Dice

Random numbers are useful for experimentation

They also appear in lots of algorithms, e.g.,

- Primality tests

Implementing Dice

Random numbers are useful for experimentation

They also appear in lots of algorithms, e.g.,

- Primality tests
- Machine learning techniques

Implementing Dice

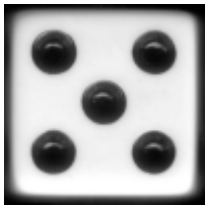
Random numbers are useful for experimentation

They also appear in lots of algorithms, e.g.,

- Primality tests
- Machine learning techniques

Implementing Dice

Random numbers are useful for experimentation



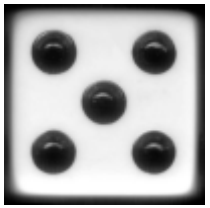
They also appear in lots of algorithms, e.g.,

- Primality tests
- Machine learning techniques

```
def make_dice(sides=6):
    seed = 1
    multiplier = pow(7, 5)
    big_prime = pow(2, 31) - 1
    def dice():
        nonlocal seed
        seed = (multiplier * seed) % big_prime
        return (sides*seed) // big_prime + 1
    return dice
```

Implementing Dice

Random numbers are useful for experimentation



They also appear in lots of algorithms, e.g.,

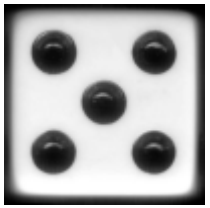
- Primality tests
- Machine learning techniques

```
def make_dice(sides=6):
    seed = 1
    multiplier = pow(7, 5)
    big_prime = pow(2, 31) - 1
    def dice():
        nonlocal seed
        seed = (multiplier * seed) % big_prime
        return (sides*seed) // big_prime + 1
    return dice
```

16807

Implementing Dice

Random numbers are useful for experimentation



They also appear in lots of algorithms, e.g.,

- Primality tests
- Machine learning techniques

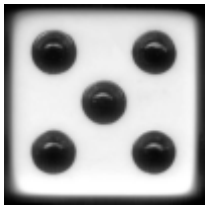
```
def make_dice(sides=6):
    seed = 1
    multiplier = pow(7, 5)
    big_prime = pow(2, 31) - 1
    def dice():
        nonlocal seed
        seed = (multiplier * seed) % big_prime
        return (sides*seed) // big_prime + 1
    return dice
```

16807

2147483647

Implementing Dice

Random numbers are useful for experimentation



They also appear in lots of algorithms, e.g.,

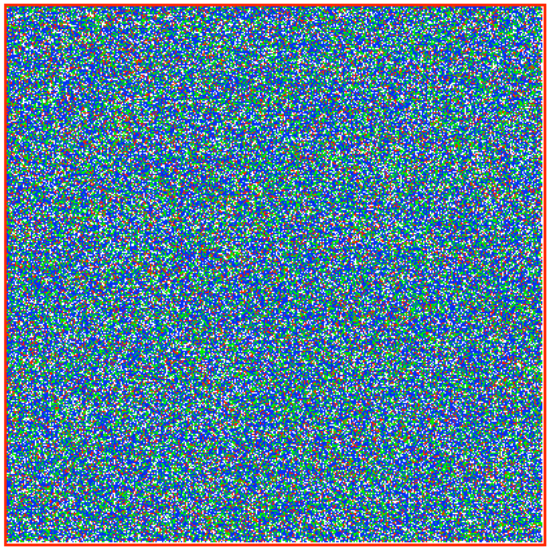
- Primality tests
- Machine learning techniques

```
def make_dice(sides=6):
    seed = 1
    multiplier = pow(7, 5)
    big_prime = pow(2, 31) - 1
    def dice():
        nonlocal seed
        seed = (multiplier * seed) % big_prime
        return (sides*seed) // big_prime + 1
    return dice
```

16807

2147483647

P1 = 16807, P2 = 0, N = 2147483647



100000 dots drawn, seed = 1

http://www.math.utah.edu/~pa/Random/Random.html

S.K. Park and K.W. Miller, " Random Number Generators: Good Ones Are Hard To Find", Communications of the ACM, October 1988, pp. 1192-1201.

Implementing a Mutable Container Object

Demo

Dispatch Functions

A technique for packing multiple behaviors into one function

Dispatch Functions

A technique for packing multiple behaviors into one function

```
def pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Dispatch Functions

A technique for packing multiple behaviors into one function

```
def pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

Dispatch Functions

A technique for packing multiple behaviors into one function

```
def pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:

Dispatch Functions

A technique for packing multiple behaviors into one function

```
def pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:

- One conditional statement with several clauses

Dispatch Functions

A technique for packing multiple behaviors into one function

```
def pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:

- One conditional statement with several clauses
- Headers perform equality tests on the message

Message Passing

Message Passing

An approach to organizing the relationship among different pieces of a program

Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

- What is your fourth element?

Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

- What is your fourth element?
- Change your third element to this new value. (please?)

Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

- What is your fourth element?
- Change your third element to this new value. (please?)

Encapsulates the behavior of all operations on a piece of data

Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

- What is your fourth element?
- Change your third element to this new value. (please?)

Encapsulates the behavior of all operations on a piece of data

Important historical role:
The message passing approach strongly influenced object-oriented programming
(next lecture)

A Mutable Container That Uses Message Passing

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':

            contents = value
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':

            contents = value



 return dispatch
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':

            contents = value

 return dispatch
```

```
def container(contents):

    def get():

        return contents

    def put(value):

        nonlocal contents

        contents = value

    return get, put
```

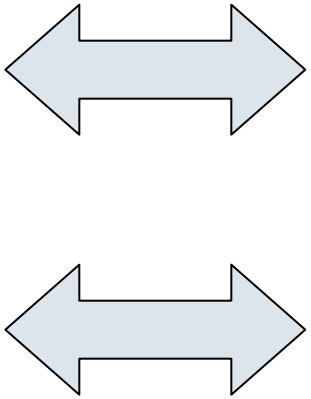
A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':

            contents = value


 return dispatch
```

```
def container(contents):


    def get():

        return contents

    def put(value):

        nonlocal contents

        contents = value

    return get, put
```

A Mutable Container That Uses Message Passing

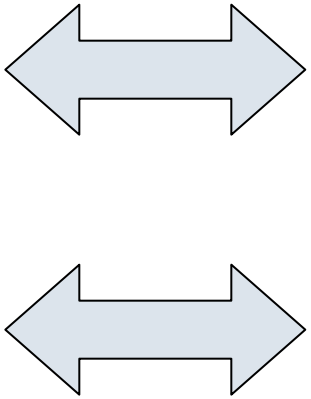
```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':

            contents = value


 return dispatch
```



```
def container(contents):


    def get():

        return contents

    def put(value):

        nonlocal contents

        contents = value

    return get, put
```

A Mutable Container That Uses Message Passing

```
def container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':

            contents = value


 return dispatch
```



```
def container(contents):


    def get():

        return contents

    def put(value):

        nonlocal contents

        contents = value

    return get, put
```

Demo

Implementing Mutable Recursive Lists

Implementing Mutable Recursive Lists

Recursive List
Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
```

Recursive List
Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

Recursive List Refresher Demo

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
```

Implementing Mutable Recursive Lists

Recursive List Refresher Demo

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
```

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

Recursive List Refresher Demo

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
```

Implementing Mutable Recursive Lists

Recursive List Refresher Demo

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
```

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
```

Recursive List Refresher Demo

Implementing Mutable Recursive Lists

Recursive List Refresher Demo

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
    return dispatch
```

Implementing Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
    return dispatch
```

Recursive List Refresher Demo

Demo

Implementing Dictionaries

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):

        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)

    return dispatch
```

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)

    return dispatch
```

Question: Do we need a nonlocal statement here?

Implementing Dictionaries

```
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)

    return dispatch
```

Question: Do we need a nonlocal statement here?

Demo

Dispatch Dictionaries

Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive

Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without writing new code

Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without writing new code

A dispatch dictionary has messages as keys and functions (or data objects) as values.

Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without writing new code

A dispatch dictionary has messages as keys and functions (or data objects) as values.

Dictionaries handle the message look-up logic; we concentrate on implementing useful behavior.

Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without writing new code

A dispatch dictionary has messages as keys and functions (or data objects) as values.

Dictionaries handle the message look-up logic; we concentrate on implementing useful behavior.

In Javascript, all objects are just dictionaries

Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without writing new code

A dispatch dictionary has messages as keys and functions (or data objects) as values.

Dictionaries handle the message look-up logic; we concentrate on implementing useful behavior.

Demo

In Javascript, all objects are just dictionaries

Example: Constraint Programming

Bonus Material

Example: Constraint Programming

Bonus Material

```
a + b = c
```

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b
```

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

- We define the relationship between quantities

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

- We define the relationship between quantities
- We provide values for the "known" quantities

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

Challenge: We want a general means of combination.

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b                    p * v = n * k * t

b = c - a
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

Challenge: We want a general means of combination.

Example: Constraint Programming

Bonus Material

```
a + b = c

a = c - b

b = c - a
```

Boltzmann's constant

```
p * v = n * k * t
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

ONE WAY Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

Challenge: We want a general means of combination.

Example: Constraint Programming

Bonus Material

```
a + b = c
a = c - b
b = c - a
```

Boltzmann's constant

```
p * v = n * k * t
9 * c = 5 * (f - 32)
```

Algebraic equations are *declarative*. They describe a relation among different quantities.

ONE WAY Python functions are *procedural*. They describe how to compute a result from a set of input arguments.

Constraint programming:

- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

Challenge: We want a general means of combination.

A Constraint Network for Temperature Conversion

Bonus Material

```
9 * celsius = 5 * (fahrenheit - 32)
```

A Constraint Network for Temperature Conversion

Bonus Material

Combination idea: All intermediate quantities have values too.

```
9 * celsius = 5 * (fahrenheit - 32)
```

A Constraint Network for Temperature Conversion

Bonus Material

Combination idea: All intermediate quantities have values too.

```
9 * celsius = 5 * (fahrenheit - 32)
```

This quantity relates directly to **celsius**

A Constraint Network for Temperature Conversion

Bonus Material

Combination idea: All intermediate quantities have values too.

u

```
9 * celsius = 5 * (fahrenheit - 32)
```

This quantity relates directly to **celsius**

A Constraint Network for Temperature Conversion

Bonus Material

Combination idea: All intermediate quantities have values too.

u

```
9 * celsius = 5 * (fahrenheit - 32)
```

This quantity relates directly to **celsius**

This quantity relates directly to **fahrenheit**

A Constraint Network for Temperature Conversion

Bonus Material

Combination idea: All intermediate quantities have values too.

u

v

```
9 * celsius = 5 * (fahrenheit - 32)
```

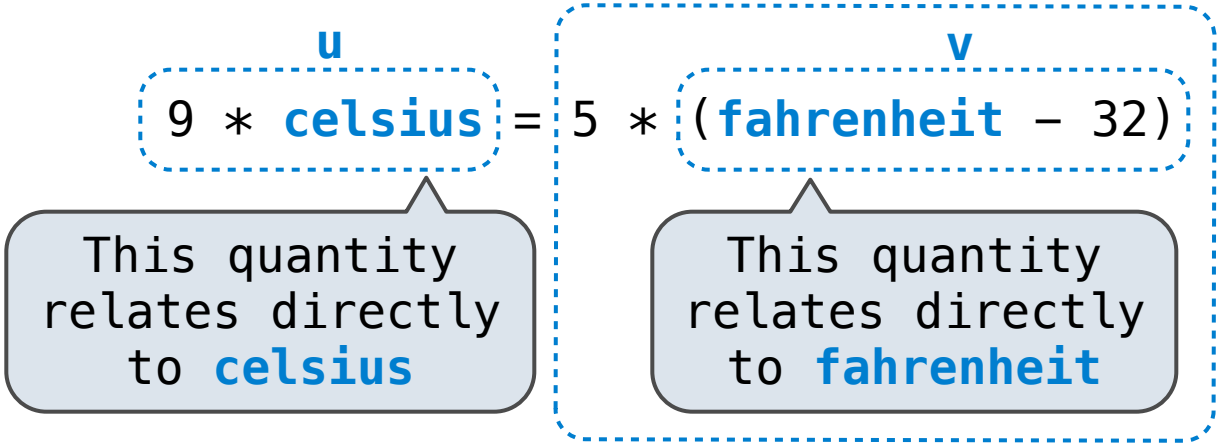
This quantity relates directly to **celsius**

This quantity relates directly to **fahrenheit**

A Constraint Network for Temperature Conversion

Bonus Material

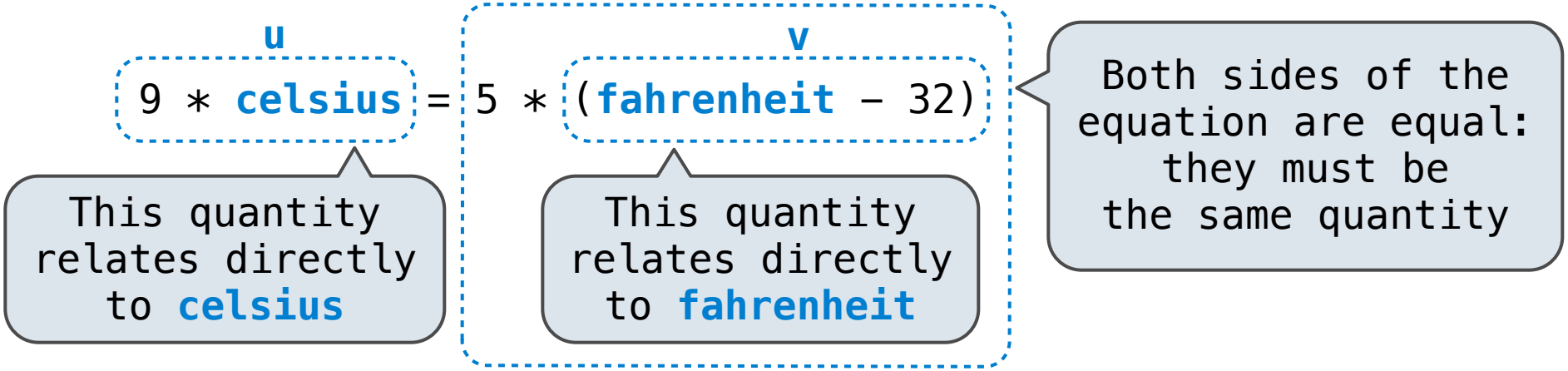
Combination idea: All intermediate quantities have values too.



A Constraint Network for Temperature Conversion

Bonus Material

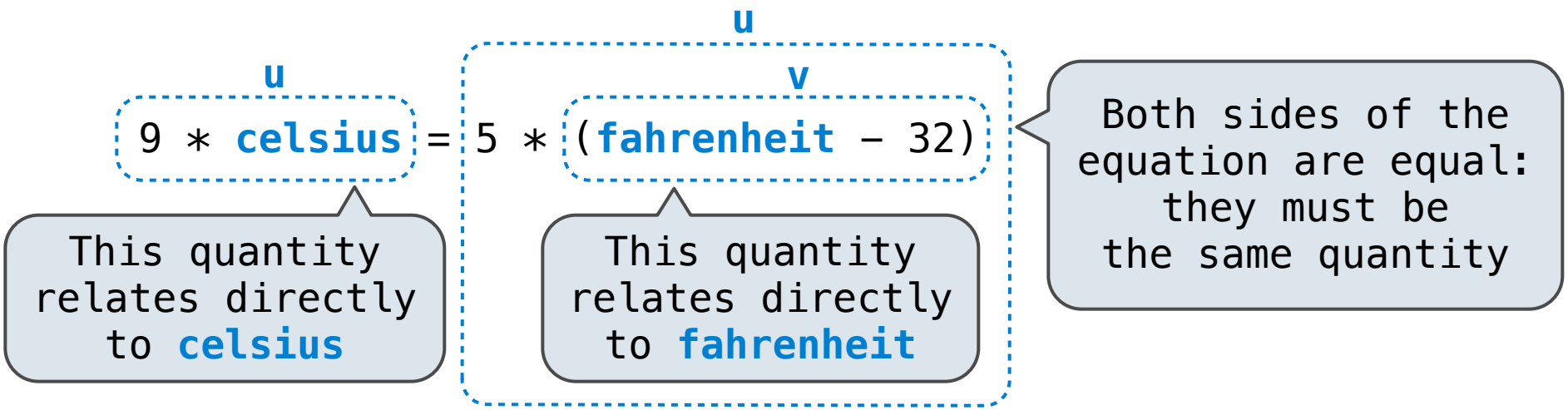
Combination idea: All intermediate quantities have values too.



A Constraint Network for Temperature Conversion

Bonus Material

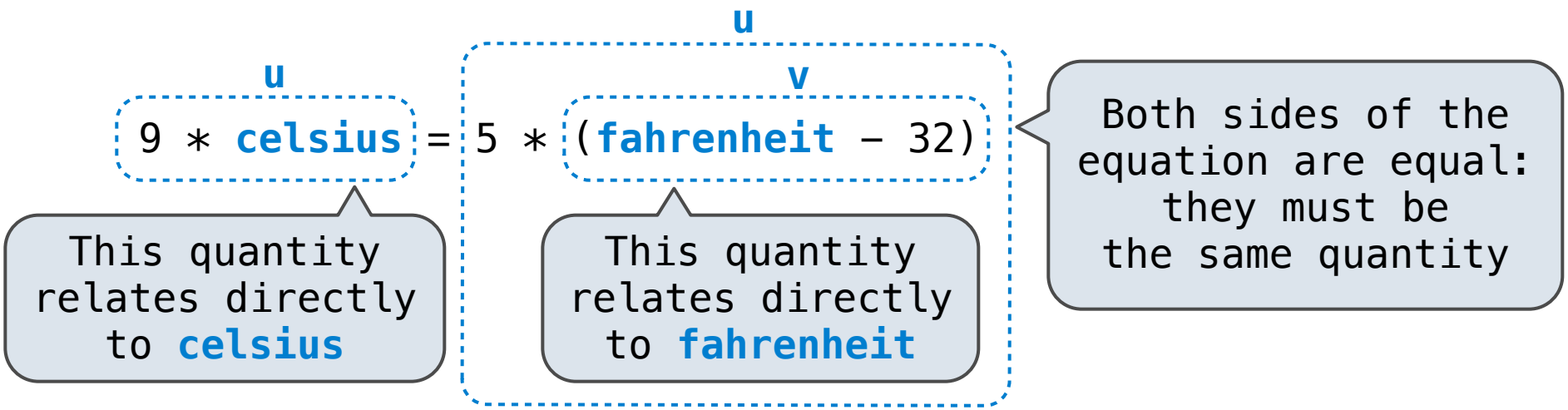
Combination idea: All intermediate quantities have values too.

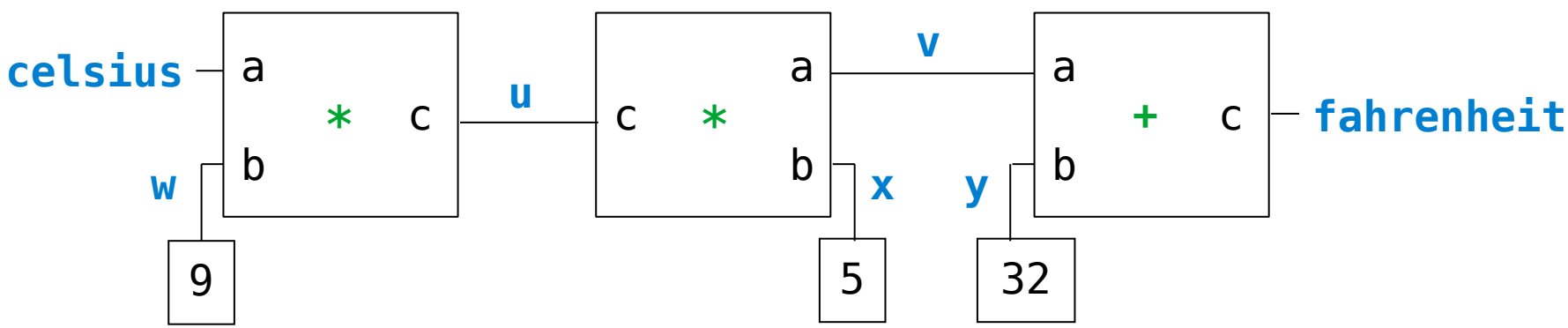


A Constraint Network for Temperature Conversion

Bonus Material

Combination idea: All intermediate quantities have values too.

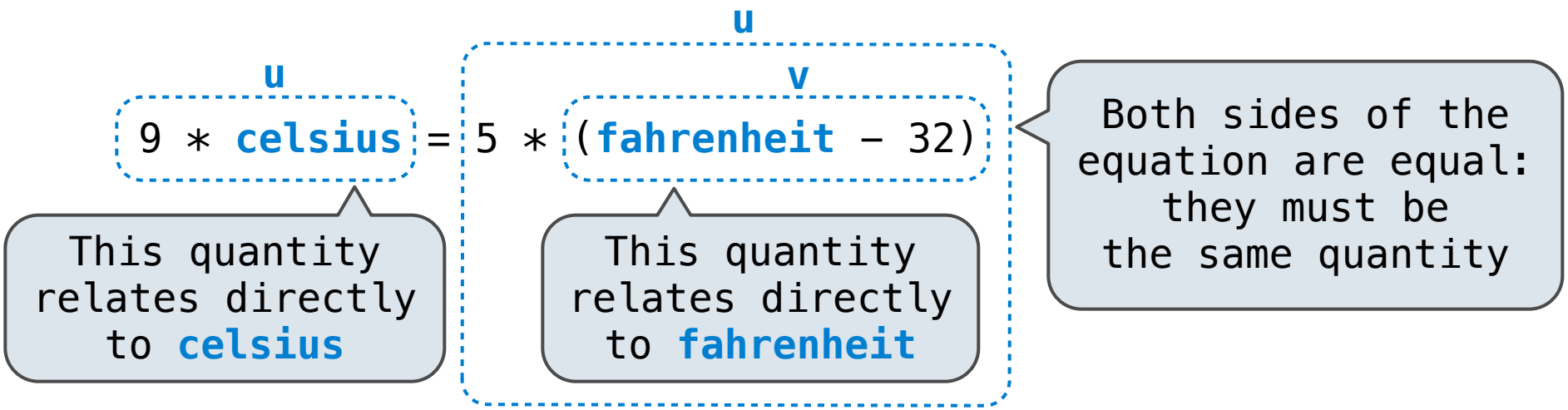


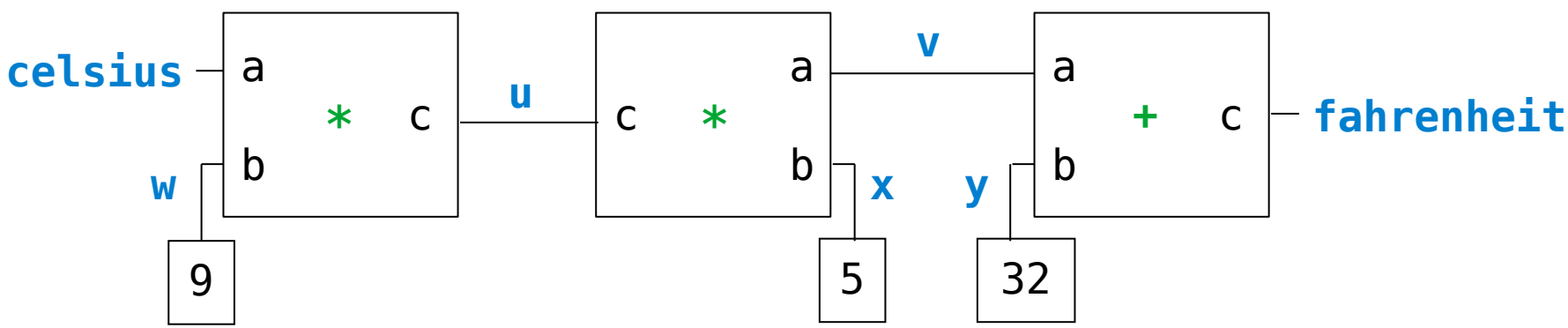


A Constraint Network for Temperature Conversion

Bonus Material

Combination idea: All intermediate quantities have values too.





Demo