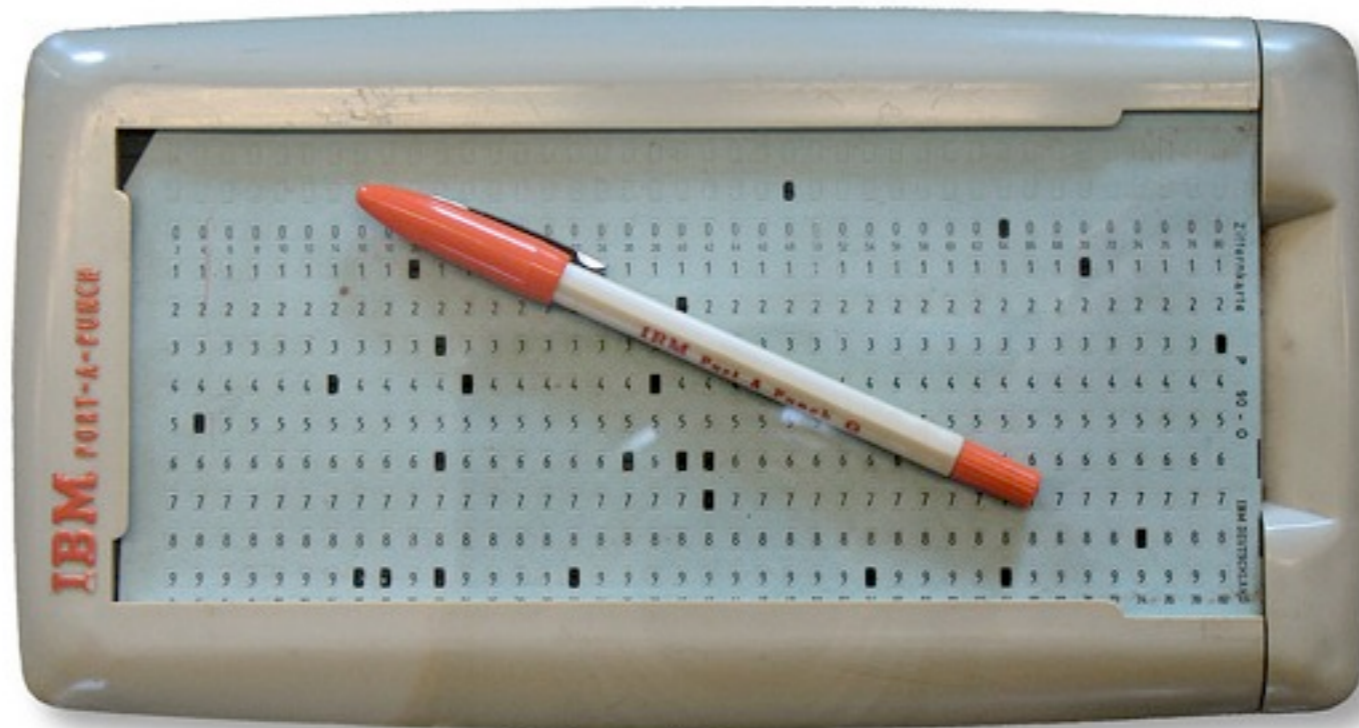# 61A Lecture 19

Wednesday, October 12

What Are Programs?

Once upon a time, people wrote programs on blackboards

Every once in a while, they would "punch in" a program

Now, we type programs as text files using editors like Emacs

Programs are just text (or cards) until we interpret them

http://en.wikipedia.org/wiki/File:IBM_Port-A-Punch.jpg

Tuesday, October 11, 2011

Evaluation rule for call expressions:

1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

Applying user-defined functions:

1. Create a new local frame that extends the environment with which the function is associated.
2. Bind the arguments to the function's formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

Execution rule for def statements:

1. Create a new function value with the specified name, formal parameters, and function body.
2. Associate that function with the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

Execution rule for assignment statements:

1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values in the first frame of the current environment.

Execution rule for conditional statements:

Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for and expressions:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

hof.p

```
    r
def i
    r
def c
    r
def s
    "
    >
    2
    "
    t
    w
    r
def p
    r
# Loc
def m
    "
    >
    >
    7
    "
    d
    r
def c
    "
    f
```

The most fundamental idea in computer science:

An *interpreter*, which determines the meaning of expressions in a programming language, is just another program.

```
    return total

def identity(k):
    return k

def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

subexpression <right>.

Designing Interpreters

All programming languages are not the same!

Common elements: User-defined functions & call expressions

Some features are often excluded: Higher-order functions, object systems, while and for statements, assignment, etc.

Coming soon: The Logo language doesn't include any of these features, but still lets us define short, powerful programs!

How can that be? Raw code is passed around like data.

Before we build interpreters:

• Recursive functions

• Recursive data structures

• Error handling

Recursive Functions

Definition: A function is called *recursive* if the body of that function calls itself, either directly or indirectly.

Implication: Executing the body of a recursive function may require applying that function again.
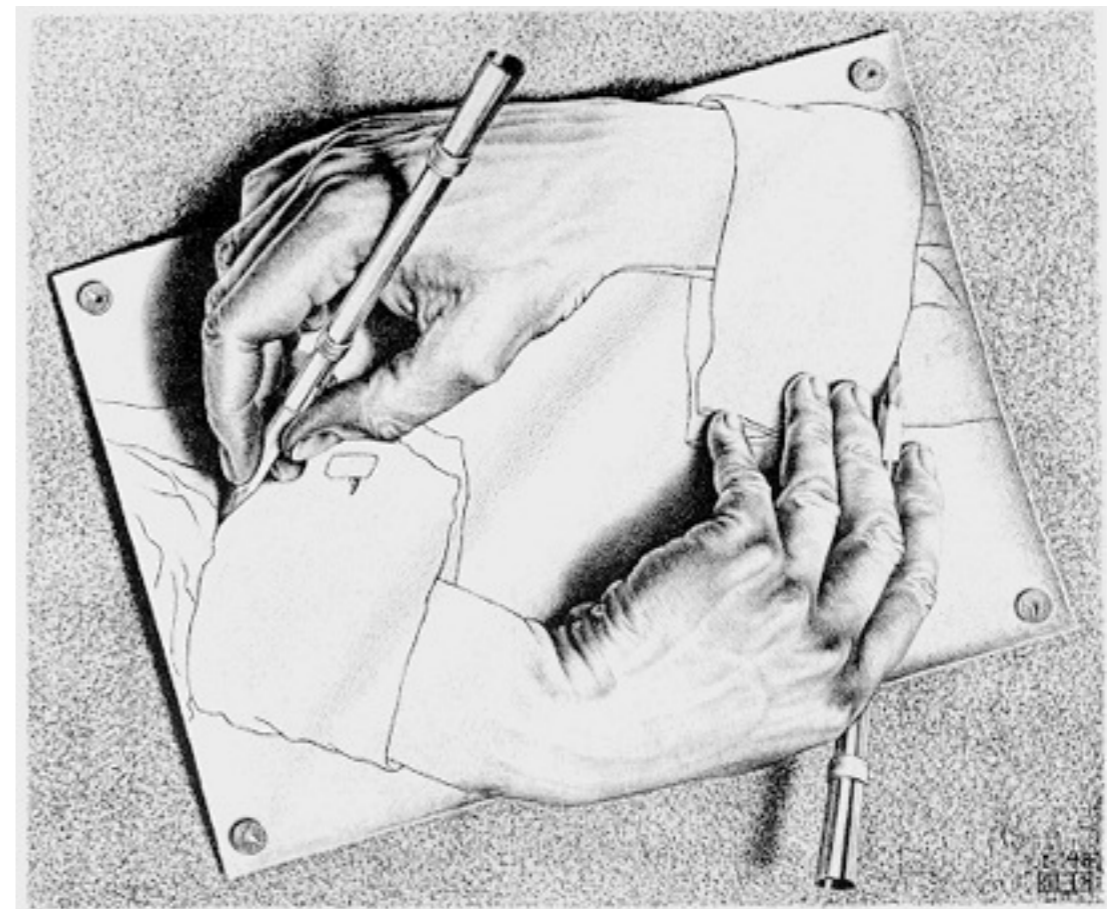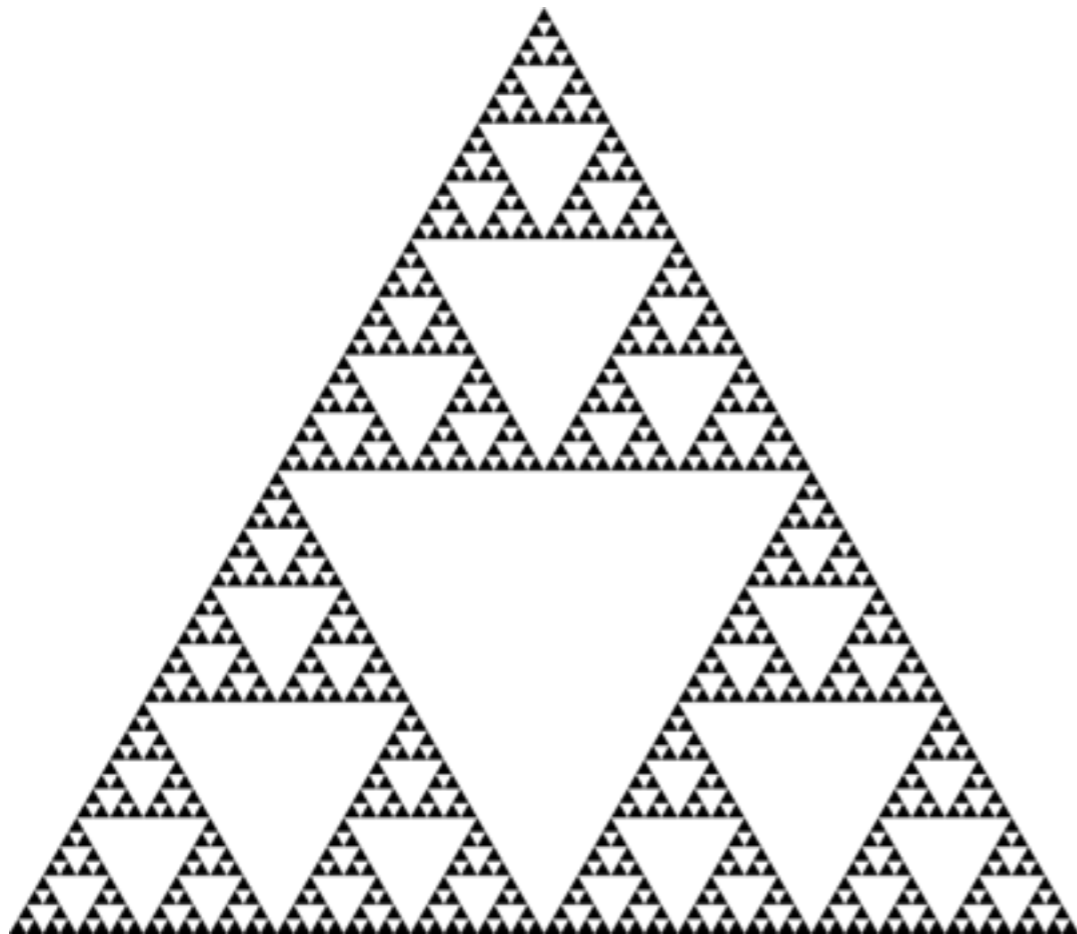

Drawing Hands, by M. C. Escher (lithograph, 1948)

Tuesday, October 11, 2011

Example: Pig Latin

Yes, you're in a college class and learning Pig Latin.

```python
def pig_latin(w):
    """Return the Pig Latin equivalent of English word w."""
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin(w[1:] + w[0])


def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```
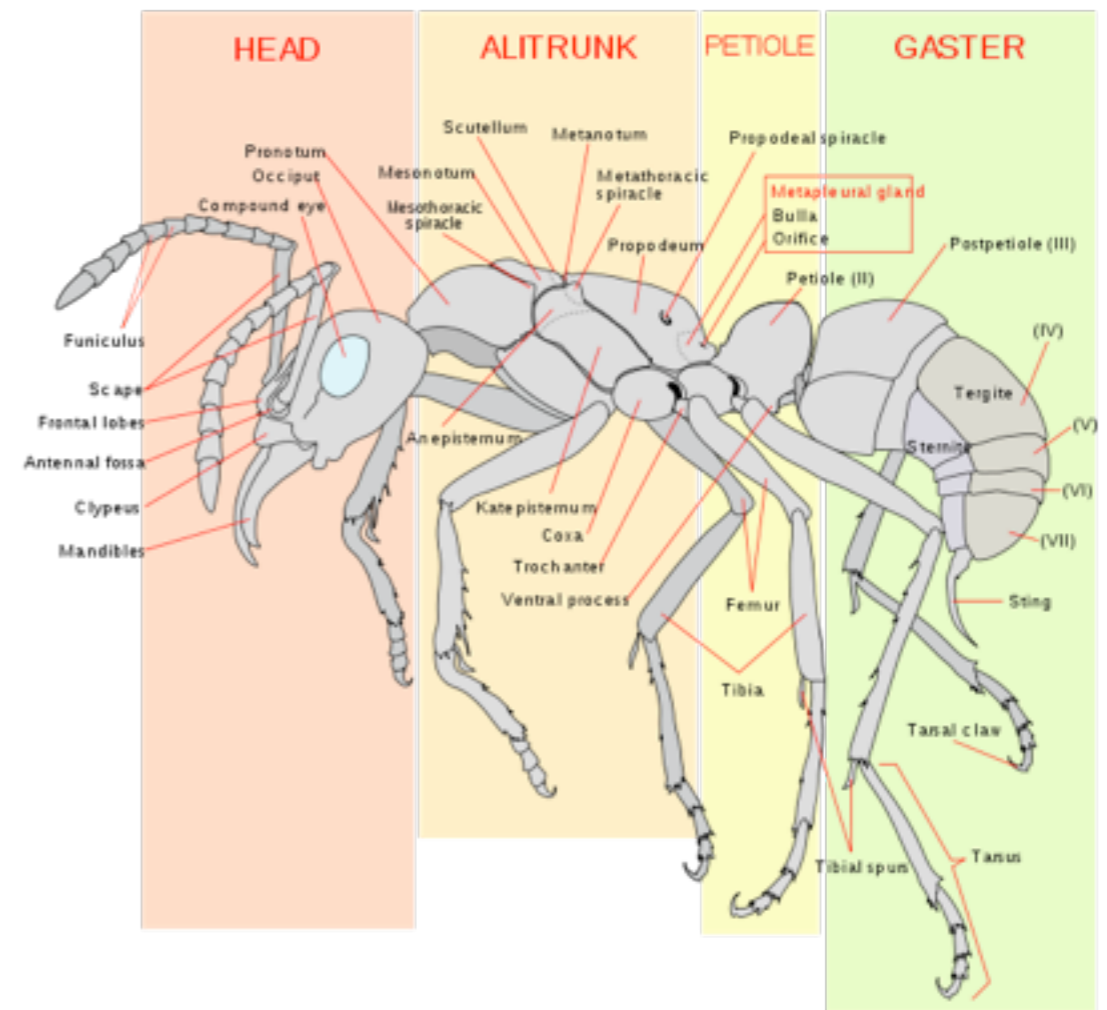
Demo

Tuesday, October 11, 2011

```
pig_latin:

starts_with_a_vowel:
```

```
pig_latin(w):
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin(w[1:] + w[0])
```

```
starts_with_a_vowel(w):
    return w[0].lower() in 'aeiou'
```

```
w: 'unp'        w: 'pun'
```

```
'unpay'
pig_latin('pun')

if starts_with_a_vowel(w):
    return w + 'ay'
return pig_latin(w[1:] + w[0])
```

```
'unpay'
pig_latin(w[1:] + w[0])

if starts_with_a_vowel(w):
    return w + 'ay'
return pig_latin(w[1:] + w[0])
```
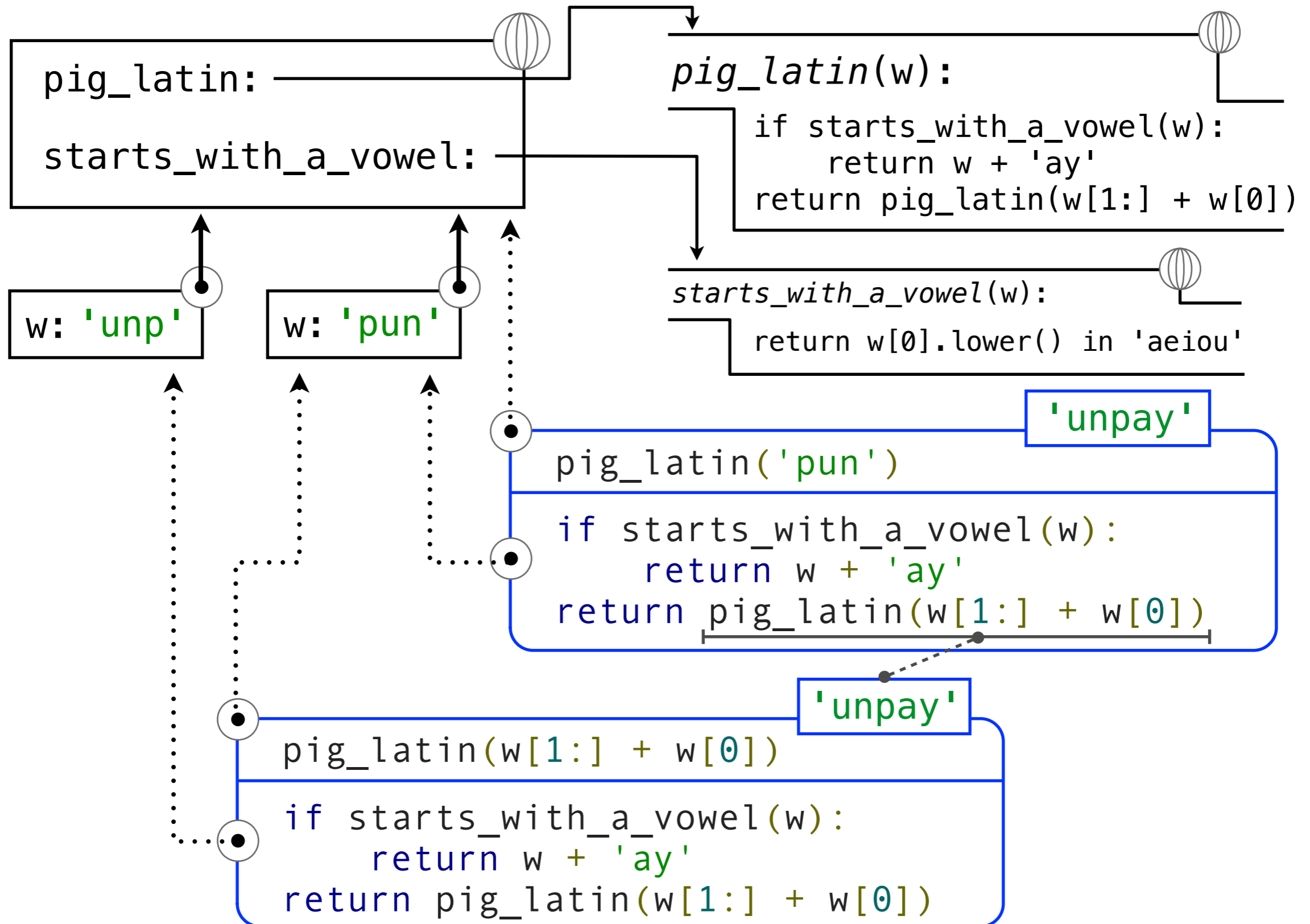
# The Anatomy of a Recursive Function

- The **def statement header** is similar to other functions

- Conditional statements check for **base cases**

- Base cases are evaluated **without recursive calls**

- Typically, all other cases are evaluated **with recursive calls**

Recursive functions are like ants (more or less)


```
def pig_latin(w):

    if starts_with_a_vowel(w):

        return w + 'ay'

    return pig_latin(w[1:] + w[0])
```

http://en.wikipedia.org/wiki/File:Scheme_ant_worker_anatomy-en.svg
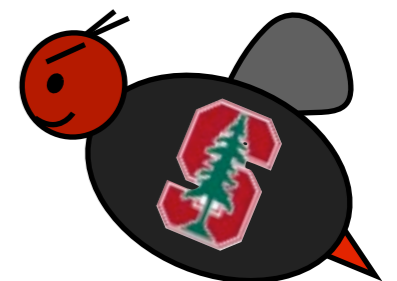
Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

```python
def fact(n):
    if n == 1:
        return 1
    return n * fact(n-1)
```

Math:

$$n! = \prod_{i=1}^{n} i$$

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n-1) & \text{otherwise} \end{cases}$$

Names:
n, total, k

n

Tuesday, October 11, 2011

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n-1)

fact(4)
```
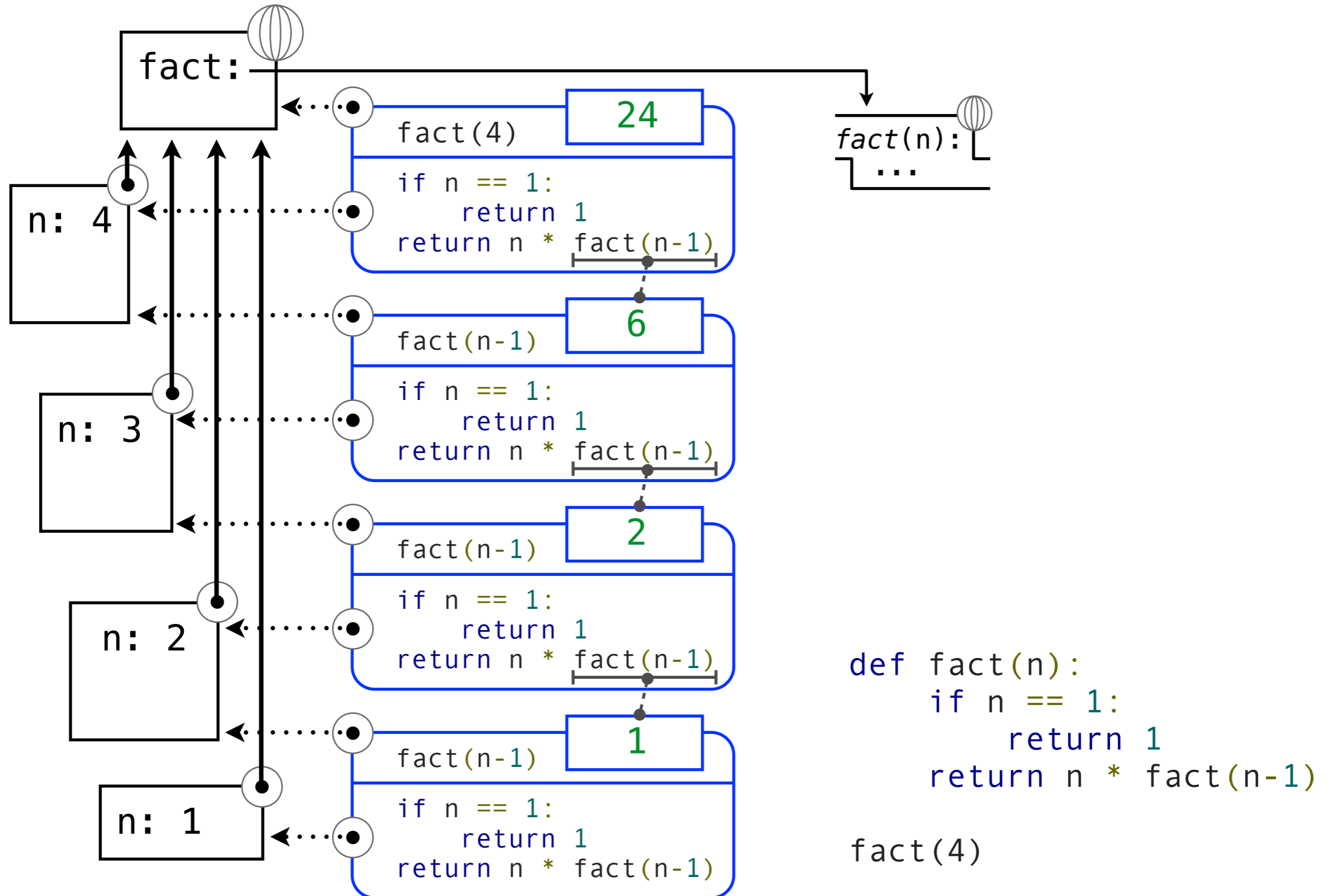
The Recursive Leap of Faith

```python
def fact(n):
    if n == 1:
        return 1
    return n * fact(n-1)
```


Is fact implemented correctly?

1. Verify the base case.

2. Treat fact(n–1) as a functional abstraction!

3. Assume that fact(n–1) is correct.

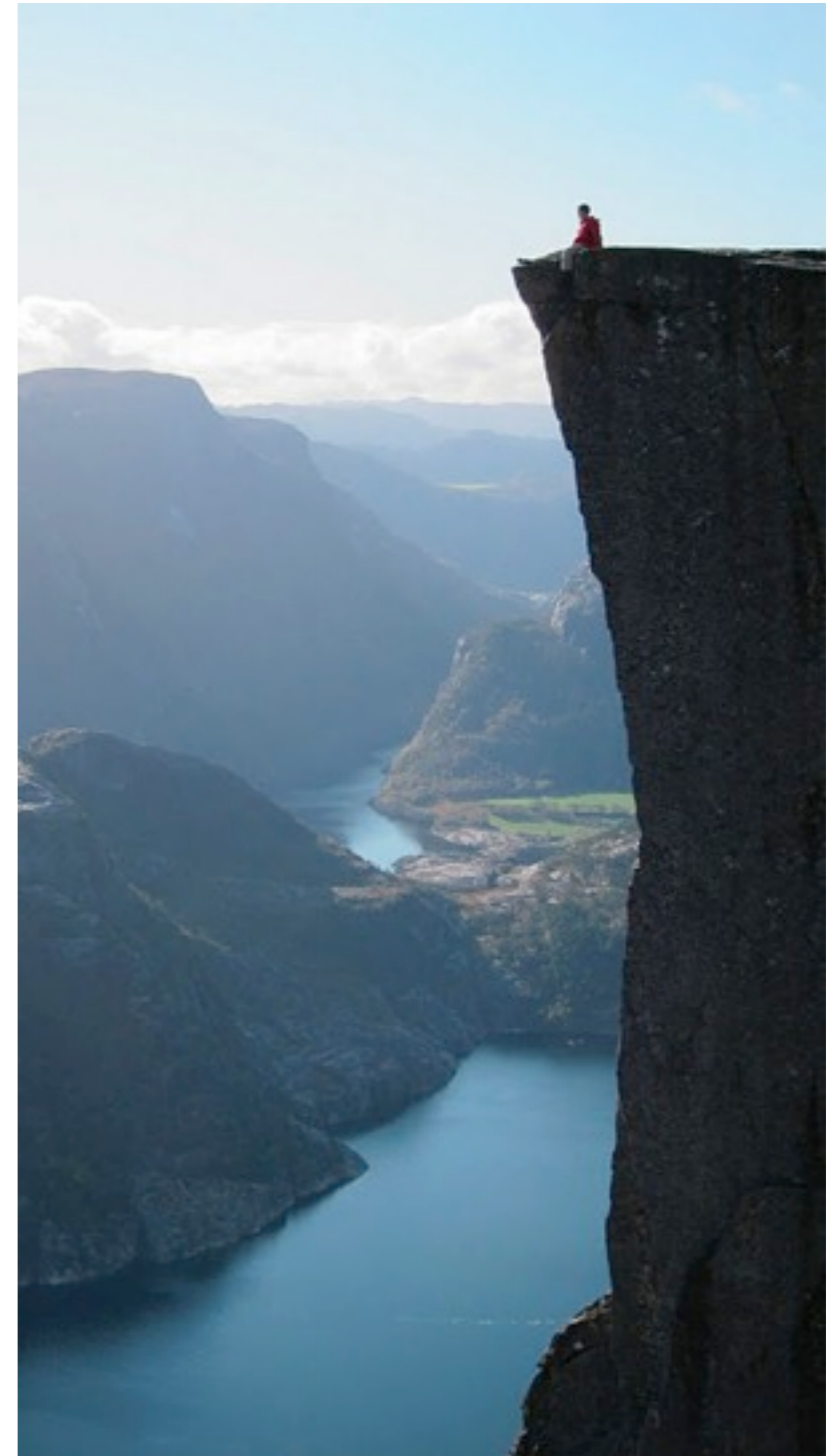4. Verify that fact(n) is correct, assuming that fact(n–1) correct.

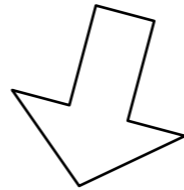Photo by Kevin Lee, Preikestolen, Norway

Example: Reverse a String

```
def reverse(s):
    """Return the reverse of a string s."""
```

Recursive idea: The reverse of a string is the reverse of the rest of the string, followed by the first letter.

antidisestablishmentarianism

a ntidisestablishmentarianism

msinairatnemhsilbatsesiditn a

```
reverse(s[1:]) + s[0]
```

Base Case: The reverse of an empty string is itself.

Tuesday, October 11, 2011

Converting Recursion to Iteration

Hard! Iteration is a special case of recursion

Idea: Figure out what state must be maintained by the function

```python
def reverse(s):
    if s == '':
        return s
    return reverse(s[1:]) + s[0]
```

> What's reversed so far?

> How to get each incremental piece

```python
def reverse_iter(s):
    r, i = '', 0
    while i < len(s):
        r, i = s[i] + r, i + 1
    return r
```

Tuesday, October 11, 2011

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion

Idea: The *state* of an iteration can be passed as parameters

```python
def reverse_iter(s):
    r, i = '', 0
    while i < len(s):
        r, i = s[i] + r, i + 1
    return r
```

Assignment becomes...

```python
def reverse2(s):
    def rev(s, r, i):
        if not i < len(s):
            return r
        return rev(s, s[i] + r, i + 1)
    return rev(s, '', 0)
```

Arguments to a recursive call

Tuesday, October 11, 2011