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.

How Are Evaluation Procedures Applied?

<table>
<thead>
<tr>
<th>Evaluation rule for call expressions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Evaluate the operator and operand subexpressions.</td>
</tr>
<tr>
<td>2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.</td>
</tr>
<tr>
<td><strong>Applying user-defined functions:</strong></td>
</tr>
<tr>
<td>1. Create a new local frame that extends the environment with which the function is associated.</td>
</tr>
<tr>
<td>2. Bind the arguments to the function's formal parameter names in that frame.</td>
</tr>
<tr>
<td>3. Execute the body of the function in the environment beginning at that frame.</td>
</tr>
<tr>
<td><strong>Execution rule for def statements:</strong></td>
</tr>
<tr>
<td>1. Create a new function value with the specified name, formal parameters, and function body.</td>
</tr>
<tr>
<td>2. Associate that function with the current environment.</td>
</tr>
<tr>
<td>3. Bind the name of the function to the function value in the first frame of the current environment.</td>
</tr>
<tr>
<td><strong>Execution rule for assignment statements:</strong></td>
</tr>
<tr>
<td>1. Evaluate the expression(s) on the right of the equal sign.</td>
</tr>
<tr>
<td>2. Simultaneously bind the names on the left to those values in the first frame of the current environment.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution rule for conditional statements:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each clause is considered in order.</td>
</tr>
<tr>
<td>1. Evaluate the header's expression.</td>
</tr>
<tr>
<td>2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.</td>
</tr>
<tr>
<td><strong>Evaluation rule for or expressions:</strong></td>
</tr>
<tr>
<td>1. Evaluate the subexpression &lt;left&gt;.</td>
</tr>
<tr>
<td>2. If the result is a true value v, then the expression evaluates to v.</td>
</tr>
<tr>
<td>3. Otherwise, the expression evaluates to the value of the subexpression &lt;right&gt;.</td>
</tr>
<tr>
<td><strong>Evaluation rule for and expressions:</strong></td>
</tr>
<tr>
<td>1. Evaluate the subexpression &lt;left&gt;.</td>
</tr>
<tr>
<td>2. If the result is a false value v, then the expression evaluates to v.</td>
</tr>
<tr>
<td>3. Otherwise, the expression evaluates to the value of the subexpression &lt;right&gt;.</td>
</tr>
<tr>
<td><strong>Evaluation rule for not expressions:</strong></td>
</tr>
<tr>
<td>1. Evaluate &lt;exp&gt;; The value is True if the result is a false value, and False otherwise.</td>
</tr>
<tr>
<td><strong>Execution rule for while statements:</strong></td>
</tr>
<tr>
<td>1. Evaluate the header's expression.</td>
</tr>
<tr>
<td>2. If it is a true value, execute the (whole) suite, then return to step 1.</td>
</tr>
</tbody>
</table>

The most fundamental idea in computer science:

An interpreter, which determines the meaning of expressions in a programming language, is just another program.
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)
Example: Pig Latin

Yes, you're in college, 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
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**

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

Recursive functions are like ants (more or less)

**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:**  \[ n! = \prod_{k=1}^{n} k \]

**Names:**  \( n, \text{total}, k, \text{fact_iter} \)  \( n, \text{fact} \)
The Recursive Leap of Faith

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
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.
Converting Recursion to Iteration

Can be tricky! 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]

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

What's reversed so far?

How to get each incremental piece?
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

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

Assignment becomes...

Arguments to a recursive call