Lecture #8: Sequences

- The term sequence refers generally to a data structure consisting of an indexed collection of values.
- That is, there is a first, second, third value (which CS types call #0, #1, #2, etc).
- A sequence may be finite (with a length) or infinite.
- As an object, it may be mutable (elements can change) or immutable.
- There are numerous alternative interfaces (i.e., sets of operations) for manipulating it.
- And, of course, numerous alternative implementations.
- Today: immutable, finite sequences, recursively defined.

A Recursive Definition

- A possible definition: A sequence consists of
  - An empty sequence, or
  - A first element and a sequence consisting of the elements of the sequence other than the first—the rest of the sequence or tail.
- The definition is clearly recursive ("a sequence consists of . . . a sequence . . . "), so let's call it an rlist for now.
- Suggests the following ADT interface:
  ```python
  def make_rlist(first, rest = empty_rlist):
      return first, rest
  def first(r):
      return r[0]
  def rest(r):
      return r[1]
  ```

Implementation With Pairs

- An obvious implementation uses two-element tuples (pairs). The result is called a linked list.
  ```python
  empty_rlist = None
  def make_rlist(first, rest = empty_rlist):
      return first, rest
  def first(r):
      return r[0]
  def rest(r):
      return r[1]
  ```

Box-and-Pointer Diagrams for Linked Lists

- Diagrammatically, one gets structures like this:
  ```python
  Q = make_rlist(5, make_rlist(3, empty_rlist))
  L = make_rlist(8, make_rlist(Q, make_rlist(empty_rlist, empty_rlist)))
  ```

From Recursive Structure to Recursive Algorithm

- The cases in the recursive definition of list often suggest a recursive approach to implementing functions on them.
- Example: length of an rlist:
  ```python
  def len_rlist(s):
      return len(sofar + 1, rest(s))
  ```
  ```python
  def len(sofar, s):
      if s == empty_rlist:
          return sofar
      else:
          return len(sofar + 1, rest(s))
  ```

Tail Recursion (Again)

- Can't directly make len_rlist iterative.
- But a slight modification makes it possible:
  ```python
  def len_rlist(s):
      sofar = 0
      while s != empty_rlist:
          sofar, s = sofar + 1, rest(s)
      return sofar
  ```
Another Example: Selection

- Want to extract item #k from an rlist (number from 0).
- Recursively:
  ```python
def getitem_rlist(s, i):
    """Return the element at index 'i' of recursive list 's'."
    >>> getitem_rlist(make_rlist(2, make_rlist(3, make_rlist(4))), 1)
    3"
    if i == 0: return first(s)
    else: return getitem_rlist(rest(s), i-1)
```

Iterative getitem_rlist

```python
def getitem_rlist(s, i):
    """Return the element at index 'i' of recursive list 's'."
    while i != 0:
        s, i = rest(s), i-1
    return first(s)
```

Applying to All Elements

- Given an rlist, I'd like to create the list of the squares of its elements:
  ```python
def square_rlist(s):
    """The list of squares of the elements of 's'."
    if s == empty_rlist:
        return empty_rlist
    else:
        return make_rlist(first(s)**2, square_rlist(rest(s)))
```

On to Higher Orders!

```python
def map_rlist(f, s):
    """The list of values f(x) for each element x of 's' in order.""
    if s == empty_rlist:
        return empty_rlist
    else:
        return make_rlist(f(first(s)), map_rlist(f, rest(s)))
```

- So `square_rlist(L)` is `map_rlist(lambda x:x**2, L)`.
- [Python 3 produces a different kind of result from its `map` function; we'll get to it.]
- Iterative version not so easy here!
Extending rlists

Joining two lists together is called "appending" in most languages. Python uses "append" to mean "add an item," and uses the term "extend" for joining lists.

def extend_rlist(left, right):
    """The sequence of items of rlist 'left'
followed by the items of 'right'."""
    if __________:
        return __________
    else:
        return __________

Extending rlists (II)

Again, iterative version is not obvious. Can you find one?