

## 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.

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 1

## 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:

```
empty_rlist = ...
def make_rlist(first, rest = empty_rlist):
    """A recursive list, r, such that first(r) is 'first' and
    rest(r) is 'rest,' which must be an rlist."""
    def first(r):
        """The first item in r."""
    def rest(r):
        """The tail of r."""
    def isempty(r):
        """True iff r is the empty sequence"""
```

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 2

## Implementation With Pairs

- An obvious implementation uses two-element tuples (pairs). The result is called a *linked list*.

```
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]
```

Last modified: Fri Mar 2 00:41:58 2012

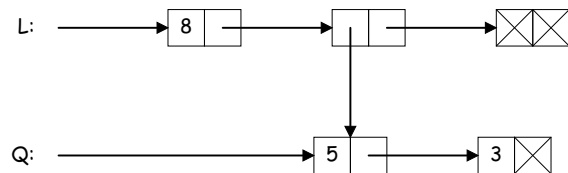
CS61A: Lecture #8 3

## Box-and-Pointer Diagrams for Linked Lists

- Diagrammatically, one gets structures like this:

```
# The sequence containing: 8; the sequence containing 5 and 3;
# and the empty sequence
Q = make_rlist(5, make_rlist(3, empty_rlist))
L = make_rlist(8,
               make_rlist(Q, make_rlist(empty_rlist, empty_rlist)))

# or
# Q = make_rlist(5, make_rlist(3))
# L = make_rlist(8, make_rlist(Q, make_rlist(empty_rlist)))
```



Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 4

## 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:

```
def len_rlist(s):
    """The length of rlist 's'."""
    if s == empty_rlist:
        return 0
    else:
        return 1 + len_rlist(rest(s))
    # A first element and
    # the rest of the list
```

- Q: Why do we know the comment is accurate?
- A: Because we assume the comment is accurate! (For "smaller" arguments, that is).
- An example of reasoning by *structural induction*...
- ...or *recursive thinking* about data structures.

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 5

## Tail Recursion (Again)

- Can't directly make *len\_rlist* iterative.
- But a slight modification makes it possible:

```
def len_rlist(s):
    def len(sofar, s):
        """'sofar' + the length of 's'"""
        if s == empty_rlist:
            return sofar
        else:
            return len(sofar + 1, rest(s))
    len(0, s)
```

- We simply return the value of the recursive call to *len* directly, so this version is *tail recursive*, and can become a loop:

```
def len_rlist(s):
    sofar = 0
    while s != empty_rlist:
        sofar, s = sofar+1, rest(s)
    return sofar
```

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 6

### Another Example: Selection

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
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))), 3)
    3

    if ____: return ____
    else:    return ____
```

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 7

### getitem\_rlist (II)

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
def getitem_rlist(s, i):
    """Return the element at index 'i' of recursive list 's'."""

    if i == 0: return first(s)
    else:     return getitem_rlist(rest(s), i-1)
```

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 8

### Iterative getitem\_rlist

```
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)
```

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 9

### Applying to All Elements

- Given an rlist, I'd like to create the list of the squares of its elements:

```
def square_rlist(s):
    """The list of squares of the elements of 's'."""
    if ____:
        return ____
    else:
        return ____
```

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 10

### Applying to All Elements (II)

- Given an rlist, I'd like to create the list of the squares of its elements:

```
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)))
```

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 11

### On to Higher Orders!

```
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!

Last modified: Fri Mar 2 00:41:58 2012

CS61A: Lecture #8 12

## 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)

- 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 left == empty_rlist:
        return right
    else:
        return make_rlist(first(left),
                           extend_rlist(rest(left), right))
```

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