

## Lecture #9: Operations on Sequences

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 1

## Another Higher-Order Operator: reduce

- We've seen a generalized way to accumulate a result in homework:

```
def accumulate(combiner, start, n, term):
    total, k = start, 1
    while k <= n:
        total, k = combiner(total, term(k)), k + 1
    return total
```

where `term` is a function.

- For sequences, the function is more conventionally named *reduce*:

```
def reduce_rlist(f, seq, start):
    """Assuming 'f' is a binary function and 'seq' an
    n-element rlist containing (e1, ..., en), returns
    f(...(f(f(start, e1), e2), ...), en)
    ('start' if n is 0)."""

    if _____: return _____
    else: return _____
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 2

## Reduce, implemented

- First, recursively.

```
def reduce_rlist(f, seq, start):
    """Assuming 'f' is a binary function and 'seq' an n-element
    rlist containing (e1, ..., en), returns
    f(...(f(f(start, e1), e2), ...), en)
    (which is 'start' if n is 0)."""
    if seq == empty_rlist: return start
    else: return reduce_rlist(f, rest(seq),
                              f(start, first(seq)))
```

- Iterative version?

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 3

## Reduce, implemented (II)

- Recursive:

```
def reduce_rlist(f, seq, start):
    """..."""
    if seq == empty_rlist: return start
    else: return reduce_rlist(f, rest(seq),
                              f(start, first(seq)))
```

- Iterative:

```
def reduce_rlist(f, seq, start):
    """Assuming 'f' is a binary function and 'seq' an n-element
    rlist containing (e1, ..., en), returns
    f(...(f(f(start, e1), e2), ...), en)
    (which is 'start' if n is 0)."""
    while seq != empty_rlist:
        seq, start = rest(seq), f(start, first(seq))
    return start
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 4

## Filtering

- Reduce and map unconditionally apply their function arguments to elements of a list. They are essentially loops.
- The analog of applying an if statement to items in a list is called *filtering*:

```
def filter_rlist(cond, seq):
    """The rlist consisting of the subsequence of
    rlist 'seq' for which the 1-argument function 'cond'
    returns a true value."""

    if _____: return _____
    elif _____: return _____
    else: return _____
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 5

## Filtering Implemented

```
def filter_rlist(cond, seq):
    """The rlist consisting of the subsequence of
    rlist 'seq' for which the 1-argument function 'cond'
    returns a true value."""

    if seq == empty_rlist:
        return empty_rlist
    elif cond(first(seq)):
        return make_rlist(first(seq),
                           filter_rlist(cond, rest(seq)))
    else:
        return filter_rlist(cond, rest(seq))
```

- Oops! Not tail-recursive. Iteration is problematic (again).
- In fact, until we get to talking about mutable recursive lists, we won't be able to do it iteratively without creating an extra list along the way.

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 6

## Reversing a List?

- As is often the case, you can easily get a recursive program about rlists by considering how first and rest are related to the result.
- For example:

```
def reverse_rlist(seq):
    """The rlist containing the items in rlist 'seq' in reverse
    order."""
    if seq == empty_rlist:
        return empty_rlist
    else:
        return extend_rlist(reverse_rlist(rest(seq)),
                             make_rlist(first(seq)))
```

- Why does this work?
- Why is it a horrendously bad implementation?

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 7

## Counting the Cost

- Each execution of `extend_rlist` creates an entirely new bunch of tuples to represent the items in the left argument.
- So, the last item in the list gets copied  $N - 1$  times, if  $N$  is the length of the list. Second-to-last  $N - 2$  times, etc.
- Thus, time to reverse is at least proportional  $(N - 1) + (N - 2) + \dots + 1 = (N^2 - N)/2$ , which seems excessive.

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 8

## A Tail-Recursive Reverse

- To make this tail-recursive, we must carry along the list we will eventually return as an argument, adding as we go.
- I claim this will do it:

```
def reverse_rlist(seq):
    def prepend_reverse(reversed_part, seq):
        """Returns the rlist consisting of the reverse of
        rlist 'seq' followed by rlist reversed_part."""
        if ____:
            return reversed_part
        else:
            return _____
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 9

## A Tail-Recursive Reverse (Filled In)

- At each step, add the next item from the sequence to the front of the reversed part (which contains the items that originally preceded it):

```
def reverse_rlist(seq):
    def prepend_reverse(reversed_part, seq):
        """Returns the rlist consisting of the reverse of
        rlist 'seq' followed by rlist reversed_part."""
        if seq == empty_rlist:
            return reversed_part
        else:
            return prepend_reverse(make_rlist(first(seq),
                                             reversed_part),
                                  rest(seq))
    return prepend_reverse(empty_rlist, seq)
```

- Iterative?

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 10

## An Iterative Reverse

- The tail-recursive version:

```
def reverse_rlist(seq):
    def prepend_reverse(reversed_part, seq):
        if seq == empty_rlist:
            return reversed_part
        else:
            return prepend_reverse(make_rlist(first(seq),
                                             reversed_part),
                                  rest(seq))
    return prepend_reverse(empty_rlist, seq)
```

- is easily made into a loop:

```
def reverse_rlist(seq):
    reversed_part = empty_rlist
    while seq != empty_rlist:
        reversed_part, seq = \
            make_rlist(first(seq), reversed_part), rest(seq)
    return reversed_part
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 11

## Filtering Done Tail-Recursively (Reversed)

- It's not too difficult to come up with a tail-recursive (and then immediately, an iterative) version of `filter_rlist`, as long as you don't mind getting the *reverse* of the desired result!

```
def filter_rlist_reverse(cond, seq):
    """The rlist consisting of the subsequence of rlist 'seq' for
    which the 1-argument function 'cond' returns a true value, in
    reverse."""
    def prepend_filter(filtered_part, seq):
        """The rlist consisting of the subsequence of rlist 'seq'
        for which 'cond' returns a true value, in reverse,
        prepended to filtered_part."""
        if ____: return ____
        elif ____: return _____
        else: return _____
    return prepend_filter(empty_rlist, seq)
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 12

## Filtering Done Tail-Recursively (Reversed)

```
def filter_rlist_reverse(cond, seq):
    """The rlist consisting of the subsequence of rlist 'seq' for
    which the 1-argument function 'cond' returns a true value, in
    reverse."""
    def prepend_filter(filtered_part, seq):
        """The rlist consisting of the subsequence of rlist 'seq' for
        which 'cond' returns a true value, in reverse, prepended to
        filtered_part."""

        if seq == empty_rlist: return filtered_part
        elif cond(first(seq)):
            return prepend_filter(make_rlist(first(seq),
                                           filtered_part),
                                rest(seq))
        else:
            return prepend_filter(filtered_part, rest(seq))
    return prepend_filter(empty_rlist, seq)
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 13

## An Iterative, Unreversed Filter

- We can apply `reverse_rlist` to get the actual result we want.
- So the final result looks like this:

```
def filter_rlist(cond, seq):
    """The rlist consisting of the subsequence of rlist 'seq' for
    which the 1-argument function 'cond' returns a true value."""

    filtered_part = empty_rlist
    while _____:
        if _____:
            _____
        else:
            _____
    return _____
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 14

## An Iterative, Unreversed Filter (Filled In)

- We can apply `reverse_rlist` to get the actual result we want.
- So the final result looks like this:

```
def filter_rlist(cond, seq):
    """The rlist consisting of the subsequence of rlist 'seq' for
    which the 1-argument function 'cond' returns a true value."""

    filtered_part = empty_rlist
    while seq != empty_rlist:
        if cond(first(seq)):
            filtered_part, seq = \
                make_rlist(first(seq), filtered_part), rest(seq)
        else:
            seq = rest(seq)
    return reverse_rlist(filtered_part)
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 15

## Python's Sequences

- Rlists are known elsewhere as *linked lists*: they are sequences with a particular choice of interface that emphasizes their recursive structure.
- Python has a much different approach to sequences built into its standard data structures, one that emphasizes their *iterative* characteristics.
- There are several different kinds of sequence embodied in the standard types: *tuples*, *lists*, *ranges*, *iterators*, and *generators*. We'll start with the first two, which are run-of-the-mill data structures.

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 16

## Sequence Features

- For this part of the course, where we emphasize computation by *construction* rather than *modification*, the interesting characteristics include:

### - Explicit Construction:

```
t = (2, 0, 9, 10, 11) # Tuple
L = [2, 0, 9, 10, 11] # List
R = range(2, 13)     # Integers 2-12.
R0 = range(13)       # Integers 0-12.
E = range(2, 13, 2)  # Even integers 2-12.
```

### - Indexing:

```
t[2] == L[2] == 9, R[2] == 4, E[2] == 6
t[-1] == t[len(t)-1] == 11
```

### - Slicing:

```
t[1:4] == (t[1], t[2], t[3]) == (0, 9, 10),
t[2:] == t[2:len(t)] == (9, 10, 11)
t[::2] == t[0:len(t):2] == (2, 9, 11)
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 17

## Sequence Iteration

- We can write compact and clearer versions of `while` loops:

```
t = (2, 0, 9, 10, 11)
s = 0
for x in t:
    s += x
>>> print(s)
32
```

- Iteration over numbers is really the same, conceptually:

```
s = 0
for i in range(1, 10):
    s += i
>>> print(s)
45
```

Last modified: Fri Mar 2 00:40:52 2012

CS61A: Lecture #9 18

## Sequences as Conventional Interfaces

- Python 3 defines `map`, `reduce`, and `filter` on sequences just as we did on `lists`.
- So to compute the sum of the even Fibonacci numbers among the first 12 numbers of that sequence, we could proceed like this:

First 20 integers:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Map fib:

```
0 1 1 2 3 5 8 13 21 34 55 89
```

Filter to get even numbers:

```
0 2 8 34
```

Reduce to get sum:

```
44
```

- ...or:

```
reduce(add, filter(iseven, map(fib, range(12))))
```

- Why is this important? Sequences are amenable to *parallelization*.

## An aside: Streams in Unix

- Many Unix utilities operate on *streams of characters*, which are sequences.
- With the help of pipes, one can do amazing things. One of my favorites:

```
tr -c -s '[:alpha:]' '\n*' < FILE | \  
sort | \  
uniq -c | \  
sort -n -r -k 1,1 | \  
sed 20q
```

which prints the 20 most frequently occurring words in `FILE`, with their frequencies, most frequent first.