Announcements

- HKN review session for Midterm 1 in 145 Dwinelle from 5-8 PM TONIGHT.
- Rooms for midterm to be assigned by login. Please watch website and Piazza.
- Please watch Piazza for news about TA review session on Monday.
- Alternative exams will be given in the labs on Wednesday.
- No labs next week. Also no Wednesday lecture.
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 rest of the elements of the sequence other than the first (its tail).

• The definition is clearly recursive ("a sequence consists of ... and a sequence ..."), so let’s call it an rlist for now.

• Suggests the following ADT interface:

```python
# The empty rlist (unique).
empty_rlist = ...
def 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: the sequence consisting of items 1, 2,..., renumbered from 0.""
```
Implementation With Pairs

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

```python
empty_rlist = None
def 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
# The sequence 1, 3, 0, 4
L = rlist(1, rlist(3, rlist(0, rlist(4, empty_rlist))))
```

![Diagram of linked list with nodes labeled 1, 3, 0, 4, showing tuple containing head and rest of list.]
Adding Dimensions

Our *rlists* can contain anything, including other *rlists*:

```python
# The sequence containing sequences (0, 1) and (2, 3)
L = rlist(rlist(0, rlist(1, empty_list)),
           rlist(rlist(2, rlist(3, empty_list)),
                 empty_rlist))
```

![Diagram of L: 0 → 1 → 2 → 3]
Recursive Lists vs. Python Tuples

- In Python, tuples are not limited to pairs.
- Could have used \((1, 3, 0, 4)\) or \(((0, 1), (2, 3))\).
- But there are advantages to rlists:
  - For tuples, \(\text{rest}(L)\) corresponds to \(L[1:]\).
  - The time and spaced required for this operation increases linearly with the length of \(L\).
  - But \(\text{rest}(L)\) on an rlist takes constant time and no additional space.
- On the other hand,
  - Computing the length or the \(k\)th element of an rlist takes time proportional to the length of the sequence,
  - But for tuples, these are constant-time operations.
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):
    # A sequence is:
    """The length of rlist S.""
    if s == empty_rlist:
        # Empty or...
        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: Recursive thinking: Because we assume the comment is accurate! (For “smaller” arguments, that is).

- Not tail recursive: can’t directly make `len_rlist` iterative.
Tail Recursion (Again)

• But a slight modification makes iteration possible:

```python
def len_rlist(s):
    def len(sofar, s):
        """Return SOFAR + the length of rlist 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:

```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, k):
    """Return the element at index K of recursive list S.
    Assumes K >= 0.
    >>> getitem_rlist(rlist(2, rlist(3, rlist(4))), 1)
    3"

    if ______:
        return _______
    else:
        return ___________________
Another Example: Selection

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

```python
def getitem_rlist(s, k):
    """Return the element at index K of recursive list S.
    Assumes K >= 0.
    >>> getitem_rlist(rlist(2, rlist(3, rlist (4))), 1)
            3"

    if k == 0:
        return __________
    else:
        return _________________
```

Another Example: Selection

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

```python
def getitem_rlist(s, k):
    """Return the element at index K of recursive list S.
    Assumes K >= 0.
    >>> getitem_rlist(rlist(2, rlist(3, rlist(4))), 1)
    3"

    if k == 0:
        return first(s)
    else:
        return ____________________________
```
Another Example: Selection

• Want to extract item \#k from an rlist (number from 0).

• Recursively:

```python
def getitem_rlist(s, k):
    """Return the element at index K of recursive list S.
    Assumes K >= 0.
    >>> getitem_rlist(rlist(2, rlist(3, rlist(4))), 1)
    3"
    
    if k == 0:
        return first(s)
    else:
        return getitem_rlist(rest(s), k-1)
```
Iterative getitem_rlist

- From the previous version:

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

- Can transform into an iterative version:

  ```python
def getitem_rlist(s, k):
    """Return the element at index K of recursive list S.
    Assumes K >= 0."""
    while ______:
      s, k = ___________
    return ________
  ```
Iterative `getitem_rlist`

- From the previous version:

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

- Can transform into an iterative version:

```python
def getitem_rlist(s, k):
    """Return the element at index K of recursive list S.
    Assumes K >= 0."""

    while k != 0:
        s, k = __________________
    return _________
```
Iterative getitem_rlist

• From the previous version:

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

• Can transform into an iterative version:

```python
def getitem_rlist(s, k):
    """Return the element at index K of recursive list S. Assumes K >= 0."""
    while k != 0:
        s, k = rest(s), k-1
    return ________
```
Iterative getitem_rlist

• From the previous version:

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

• Can transform into an iterative version:

```python
def getitem_rlist(s, k):
    """Return the element at index K of recursive list S. Assumes K >= 0."""

    while k != 0:
        s, k = rest(s), k-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 rlist S."""
    if ____________:
        return ____________:
    else:
        return ________________________________
```
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 rlist S."""
    if s == empty_rlist:
        return ___________
    else:
        return ________________
```
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 rlist S."""
    if s == empty_rlist:
        return empty_rlist:
    else:
        return __________________________
```
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 rlist S.""
    if s == empty_rlist:
        return empty_rlist:
    else:
        return rlist(first(s)**2, square_rlist(rest(s)))
```
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 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 difficult 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.

```python
def extend_rlist(left, right):
    """The sequence of items of rlist ‘left’ followed by the items of ‘right’.""
    if ____________:
        return ______
    else:
        return ________________________________
```

• Again, iterative version is difficult.
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.

```python
def extend_rlist(left, right):
    """The sequence of items of rlist 'left'
followed by the items of 'right'.""
    if left == empty_rlist:
        return _____
    else:
        return ____________________________
```

- Again, iterative version is difficult.
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.

```python
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 ______________________________
```

• Again, iterative version is difficult.
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.

```python
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 rlist(first(left), extend_rlist(rest(left), right))
```

- Again, iterative version is difficult.
Reversing

- Given a sequence represented by an rlist L, how can I create the reverse sequence, \textit{reverse\_rlist}(L)?

\[ L = \text{rlist}(1, \text{rlist}(3, \text{rlist}(0, \text{rlist}(4, \text{empty\_rlist})))) \]

\[ R = \text{reverse\_rlist}(L) \]

\[ L: \quad 1 \quad 3 \quad 0 \quad 4 \]

\[ R: \quad 4 \quad 0 \quad 3 \quad 1 \]

- What is the reverse of \textit{empty\_rlist}? ________.

- Given an rlist L, what is the relationship between \textit{first}(L), \textit{rest}(L), and \( R = \text{reverse\_rlist}(L) \)?

\[ \text{first}(L) = \text{rest}(R) \]

\[ \text{rest}(L) = \text{first}(R) \]
Reversing

- Given a sequence represented by an `rlist L`, how can I create the reverse sequence, `reverse_rlist(L)`?

  \[L = rlist(1, rlist(3, rlist(0, rlist(4, empty_rlist))))\]
  \[R = reverse_rlist(L)\]

\[
\begin{array}{c}
L: & 1 & \rightarrow & 3 & \rightarrow & 0 & \rightarrow & 4 \\
R: & 4 & \rightarrow & 0 & \rightarrow & 3 & \rightarrow & 1
\end{array}
\]

- What is the reverse of `empty_rlist`? `empty_rlist`.

- Given an `rlist L`, what is the relationship between `first(L)`, `rest(L)`, and `R=reverse_rlist(L)`?

  \[\text{Relationship}\]
Reversing

- Given a sequence represented by an rlist $L$, how can I create the reverse sequence, $\text{reverse\_rlist}(L)$?

$L = \text{rlist}(1, \text{rlist}(3, \text{rlist}(0, \text{rlist}(4, \text{empty\_rlist}))))$

$R = \text{reverse\_rlist}(L)$

$\begin{align*}
L: & \quad \quad \quad 1 \quad \quad \quad 3 \quad \quad \quad 0 \quad \quad \quad 4 \\
R: & \quad \quad \quad 4 \quad \quad \quad 0 \quad \quad \quad 3 \quad \quad \quad 1
\end{align*}$

- What is the reverse of $\text{empty\_rlist}$? $\text{empty\_rlist}$.

- Given an rlist $L$, what is the relationship between $\text{first}(L)$, $\text{rest}(L)$, and $R = \text{reverse\_rlist}(L)$?

\[
R = \text{extend\_rlist}(\text{reverse\_rlist}(\text{rest}(L)), \text{rlist}(\text{first}(L), \text{empty\_rlist}))
\]
Iterative Reversing

• The iterative version of \texttt{rlist\_reverse} is actually not bad.
• Rlists are most conveniently build from the end (because a tuple, once created, can’t be changed).
• The \textit{last} item of a reversed list is the \textit{first} item of the original list.
• This leads to the following tail recursion:

\begin{verbatim}
def reverse_rlist(L):
    def reverse_extend(to_do, already_done):
        """The result of extending ALREADY_DONE with the reverse of TO_DO.""
        if to_do == empty_rlist:
            return empty_rlist
        else:
            return reverse_extend(rest(to_do),
                                  rlist(first(to_do), already_done))

    reverse_extend(L, empty_rlist)
\end{verbatim}

• Iterative version?