Building Recursive Structures

• In Lecture #9, we defined `map_rlist` and `filter_rlist`:

```python
def map_rlist(f, s):
    """The rlist of values F(x) for each element x of rlist S (same order).""
    if isempty(s):
        return empty_rlist
    else:
        return make_rlist(f(first(s)), map_rlist(f, rest(s)))

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

• In both cases, the original input rlist is preserved and a new list created: the operation is non-destructive.

• We’ve treated rlists as immutable: unchanging once created.
Another Example: Concatenating Rlists

• To keep with Python terminology, adding one element to the end of a list is appending, and concatenating two lists together is extending.

L1 = make_rlist(1, make_rlist(2, empty_rlist))
L2 = make_rlist(3, make_rlist(4, make_rlist(5, empty_rlist)))
L3 = extend_rlist(L1, L2)
def extend_rlist(left, right):
    """The sequence of items of rlist LEFT followed by the items of RIGHT."""

    if ________________________:
        return ________________________

    elif ________________________:
        return ________________________

    else:
        return ________________________

Last modified: Sun Feb 19 17:03:49 2017
def extend_rlist(left, right):
    """The sequence of items of rlist LEFT followed by the items of RIGHT."""
    if isempty(left):
        return right
    elif isempty(right):
        return left
    else:
        return __________________________
def extend_rlist(left, right):
    """The sequence of items of rlist LEFT followed by the items of RIGHT."""

    if isempty(left):
        return right
    elif isempty(right):
        # Not really needed
        return left
    else:
        return make_rlist(first(left),
                           extend_rlist(rest(left), right))

• Here, the left argument gets duplicated, but with its last rest value being right instead of empty_rlist.

• We could exclude the first elif clause without affecting correctness [why?]...

• ...but there is a potential advantage to having it [what?].
Still Another Example: Replacing a Leaf of a Tree

- From lecture #10, a tree's recursive structure is:
  - A label and
  - Zero or more children, each a tree.
- Example: replacing a leaf with a tree. Replacing leaf 4 on the left with the middle tree gives the tree on the right.

```python
def replace_leaf(T1, v, T2):
    """The tree T1 with any leaf whose label is V replaced by subtree T2."""
```
Replacing a Leaf of a Tree (II)

- Example: replacing a leaf with a tree. Replacing leaf 4 on the left with the middle tree gives the tree on the right.

```
def replace_leaf(T1, v, T2):
    """The tree T1 with any leaf whose label is V replaced by subtree T2."""

    if _________________________________:
        return __________________________

    else:
        return __________________________
```
Replacing a Leaf of a Tree (III)

- Example: replacing a leaf with a tree. Replacing leaf 4 on the left with the middle tree gives the tree on the right.

```python
def replace_leaf(T1, v, T2):
    """The tree T1 with any leaf whose label is V replaced by subtree T2."""
    if isleaf(T1) and label(T1) == v:
        return T2
    else:
        return make_tree(label(T1),
                         [replace_leaf(c, v, T2) for c in branches(T1)])
```

Last modified: Sun Feb 19 17:03:49 2017
Immutability and Nondestructive Operations

- The functions in this lecture (and in previous ones) did not modify existing list or tree structures (only local variables).

- That is, they were non-destructive; they preserved the original input data:

  ```python
  >>> L0 = make_rlist(-3, make_rlist(-2, make_rlist(-1)))
  >>> L0
  (-3, (-2, (-1, None)))  # Assumes empty_rlist is None.
  >>> L1 = map_rlist(abs, L0)
  >>> L1
  (3, (2, (1, None)))
  >>> L0
  (-3, (-2, (-1, None)))
  ```

- Indeed, the rlist interface makes them immutable.

- This is a very useful property:
  - List values behave like integer values (e.g.): stay around as long as needed in a computation.
  - Safe to share sublists or subtrees in two different structures.
Mutability and Destructive Operations

• What if we don’t need the original data? Then nondestructive operations have memory costs, possibly time costs as well.

• For example, in the preceding extend_rlist example, we could simply keep the same rlist objects as before, without copying anything, and just changed the pointer at the end of the left list with a pointer to the right list:

L1 = make_rlist(1, make_rlist(2, empty_rlist))
L2 = make_rlist(3, make_rlist(4, make_rlist(5, empty_rlist)))
L3 = dextend_rlist(L1, L2)  # Destructive extend
Mutating Operations

- Suppose we add two more operations to rlist:

```python
def set_first(r, v):
    """Cause first(R) to be V.""

def set_rest(r, V):
    """Cause rest(R) to be V.""
```
Destructive Extending

```python
def extend_rlist(left, right):
    """The sequence of items of rlist LEFT followed by the items of RIGHT."""
    if isempty(left):
        return right
    elif isempty(right):
        return left
    else:
        return make_rlist(first(left),
                           extend_rlist(rest(left), right))

def dextend_rlist(left, right):
    """Returns result of extending LEFT with RIGHT. May destroy original
    list LEFT."""
    if isempty(left):
        return right
    elif isempty(right):
        return left
    else:
        return _______________

    return _______________
```

Last modified: Sun Feb 19 17:03:49 2017 CS61A: Lecture #11 13
Destructive Extending (II)

```python
def extend_rlist(left, right):
    """The sequence of items of rlist LEFT followed by the items of RIGHT."""
    if isempty(left):
        return right
    elif isempty(right):
        return left
    else:
        return make_rlist(first(left),
                           extend_rlist(rest(left), right))

def dextend_rlist(left, right):
    """Returns result of extending LEFT with RIGHT. May destroy original list LEFT."""
    if isempty(left):
        return right
    elif isempty(right):
        return left
    else:
        set_rest(left, dextend_rlist(rest(left), right))
        return left
```

Last modified: Sun Feb 19 17:03:49 2017
def dmap_rlist(f, s):
    """The rlist of values F(x) for each element x of rlist S in order. May modify S."""
    if isempty(s):
        return empty_rlist  # This case doesn’t change
    else:
        ?
Destructive Mapping (II)

def dmap_rlist(f, s):
    """The rlist of values F(x) for each element x of rlist S in order. May modify S.""
    if isempty(s):
        return empty_rlist  # This case doesn’t change
    else:
        set_first(s, f(first(s)))
        dmap_rlist(f, rest(s))
    return s

>>> L0 = make_rlist(-3, make_rlist(-2, make_rlist(-1)))
>>> L0
(-3, (-2, (-1, None)))  # Assumes empty_rlist is None.
>>> L1 = dmap_rlist(abs, L0)
>>> L1
(3, (2, (1, None)))
>>> L0
(3, (2, (1, None)))  # Original data lost
def dmap_rlist2(f, s):
    """The rlist of values F(x) for each element x of rlist S in order. May modify S."""
    p = s
    while not isempty(p):
        
        
        
        return 

Iterative Version of dmap_rlist
def dmap_rlist2(f, s):
    """The rlist of values F(x) for each element x of rlist S in order. May modify S."""
    p = s
    while not isempty(p):
        set_first(p, f(first(p)))
        p = rest(p)
    return s
• Good idea to have a mental picture of the differences here.

L0 = make_rlist(-3, make_rlist(-2, make_rlist(-1)))
L1 = map_rlist(abs, L0)

L0:
-1
-2
-3

L1:
1
2
3

L2 = dmap_rlist(lambda x: x**2, L0)

L0:
1
4
9

L1:
1
2
3
Identity

• We distinguish between *identity* of objects:

\[ S_0 = (1, 2, 3); \quad S_1 = (1, 2, 3) \]

\[ (S_0 \text{ is } S_1) \implies \text{False} \]

• And *equality of contents*:

\[ (S_0 \equiv S_1) \implies \text{True} \]

• When dealing with immutable objects, we generally ignore identity; only equality of contents ever matters, and once equal always equal.

• Allows *referential transparency*: if \( S[0] \equiv 3 \), and \( S \) as a whole is not re-assigned, can substitute 3 for \( S[0] \) anywhere.

• When dealing with mutable structures, identity matters, and we don’t have referential transparency.
Identity (II)

```python
>>> S0 = [1, 2]
>>> S1 = [1, 2]
>>> S2 = S0
>>> S0 == S2 == S1
True

>>> S0[0] = 3  # Not possible with tuples
>>> S0 is S2 and S0 == S2
True

>>> S0 == S1
False

>>> S1 == S2
False
```