Lecture #11: Immutable and Mutable Data

Building Recursive Structures

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

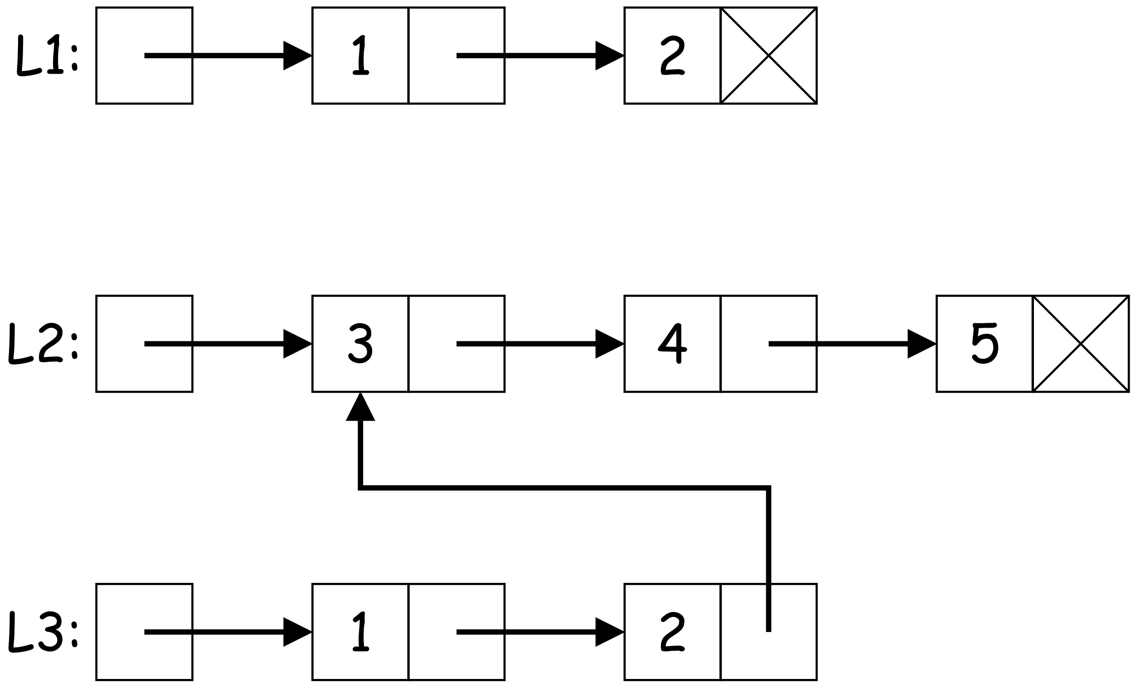
```
def map_rlist(f, s):
    """The rlist of values F(x) for each element x of rlist S (same order)."""
    if isempty(s):
        return empty_rlist
    else:
        return make_rlist(f(first(s)), map_rlist(f, rest(s)))

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)
```



Concatenating Rlists

```
def extend_rlist(left, right):
    """The sequence of items of rlist LEFT followed by the items of RIGHT."""

    if _________________________:

        return _________________________

    elif _________________________:

        return _________________________

    else:

        return __________________________________________________
```

Concatenating Rlists (II)

```
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 ________________________________________
```

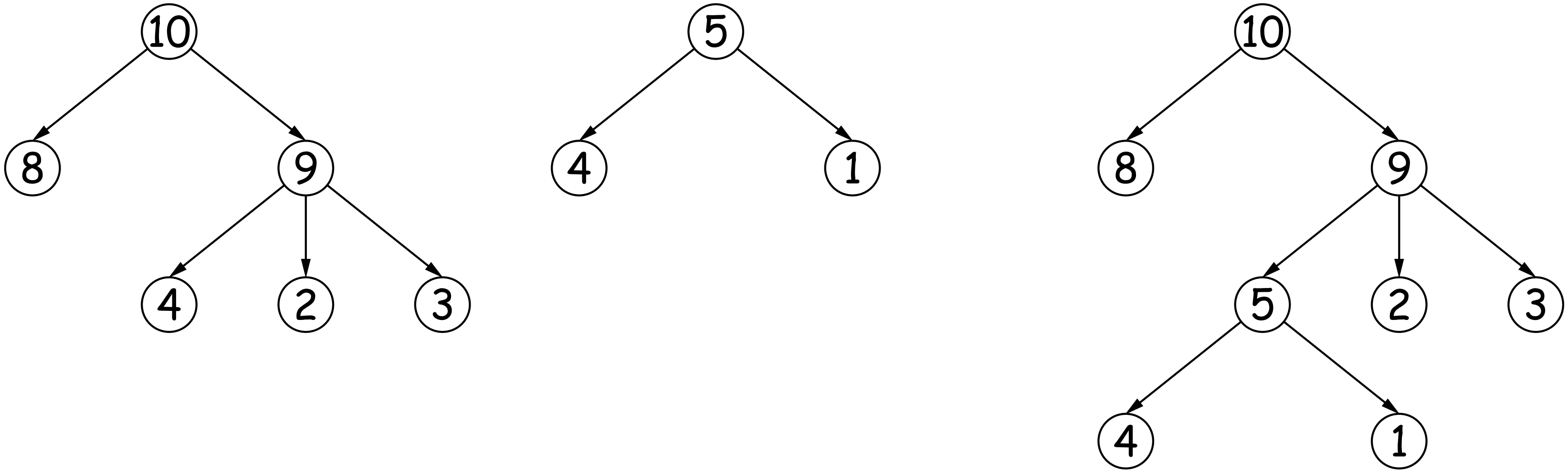
Concatenating Rlists (III)

```
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

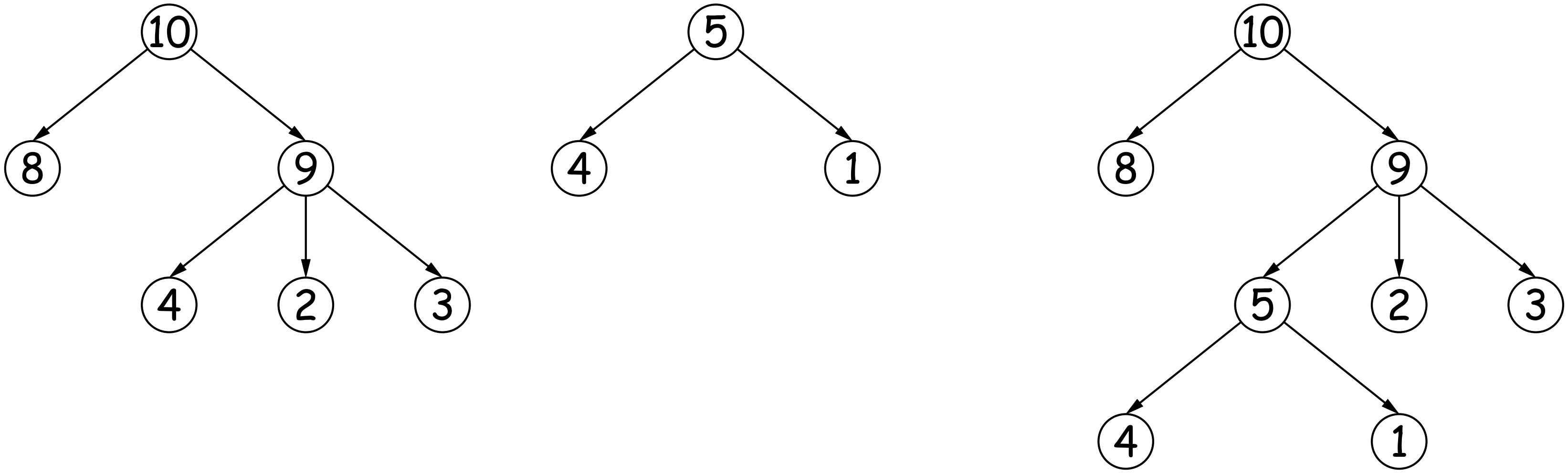
- From lecture #10, a tree's recursive structure is:
  - A label and
  - Zero or more children, each a tree.
- Example: replacing a leaf with a tree. Replacing leaf 4 on the left with the middle tree gives the tree on the right.



```
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)

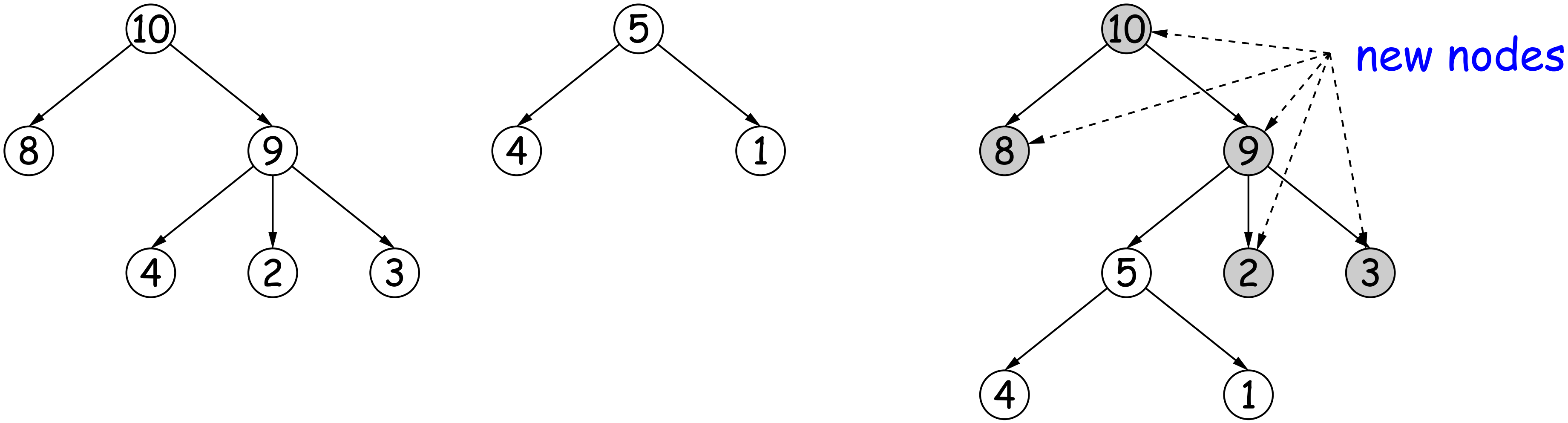
- Example: replacing a leaf with a tree. Replacing leaf 4 on the left with the middle tree gives the tree on the right.



```
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.



```
def replace_leaf(T1, v, T2):
    """The tree T1 with any leaf whose label is V
    replaced by subtree T2."""
                                         # If v is NOT in T1,
    if isleaf(T1) and label(T1) == v:    # where's the base case??!!!
         return T2
    else:
         return make_tree(label(T1),
                          [replace_leaf(c, v, T2) for c in branches(T1)])
```

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:

```
>>> 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)))
```

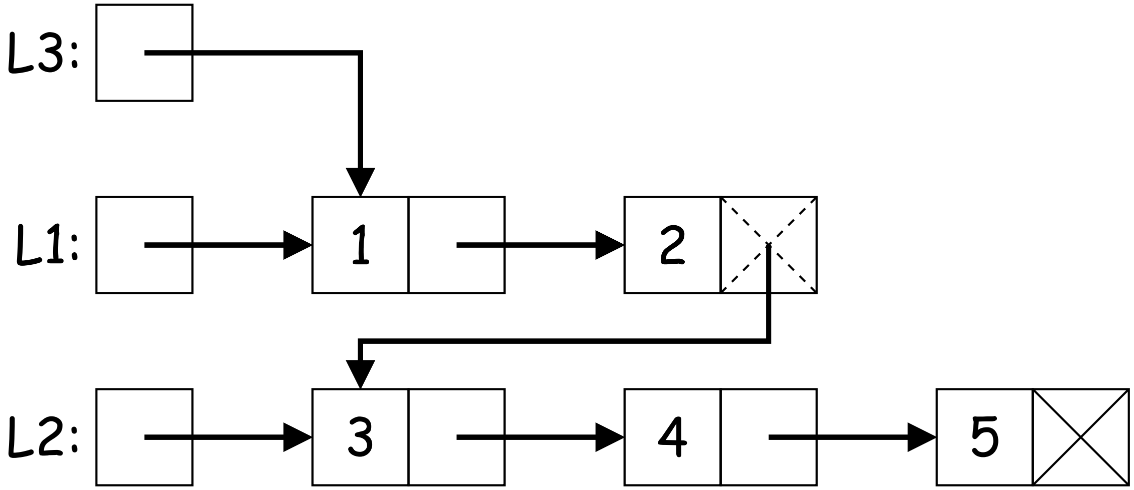
- Indeed, the `rlist` interface makes them *immutable*.
- This is a very useful property:
  - List values behave like integer values (e.g.): stay around as long as needed in a computation.
  - Safe to *share* sublists or subtrees in two different structures.

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*:

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

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

Destructive Extending

```
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 ________________
```

Destructive Extending (II)

```
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
```

Destructive Mapping

```
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
```

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):
        _________________________________
        _________________________________
    return _________________________
```

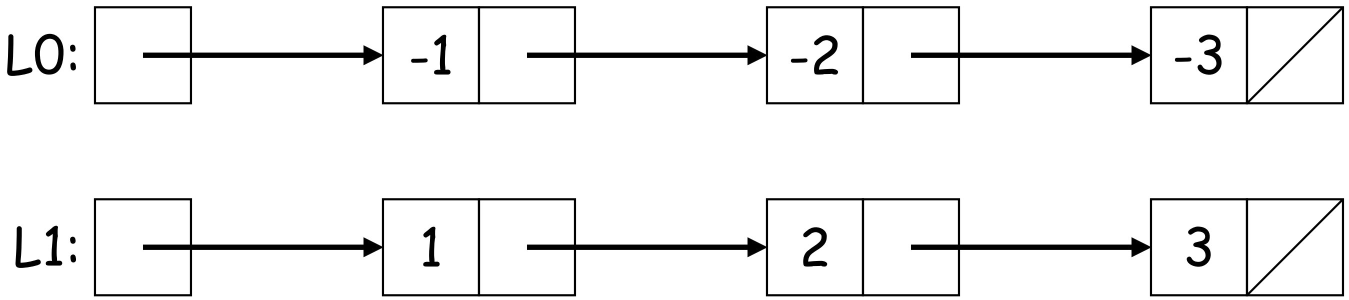
Iterative Version of dmap_rlist (II)

```
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
```

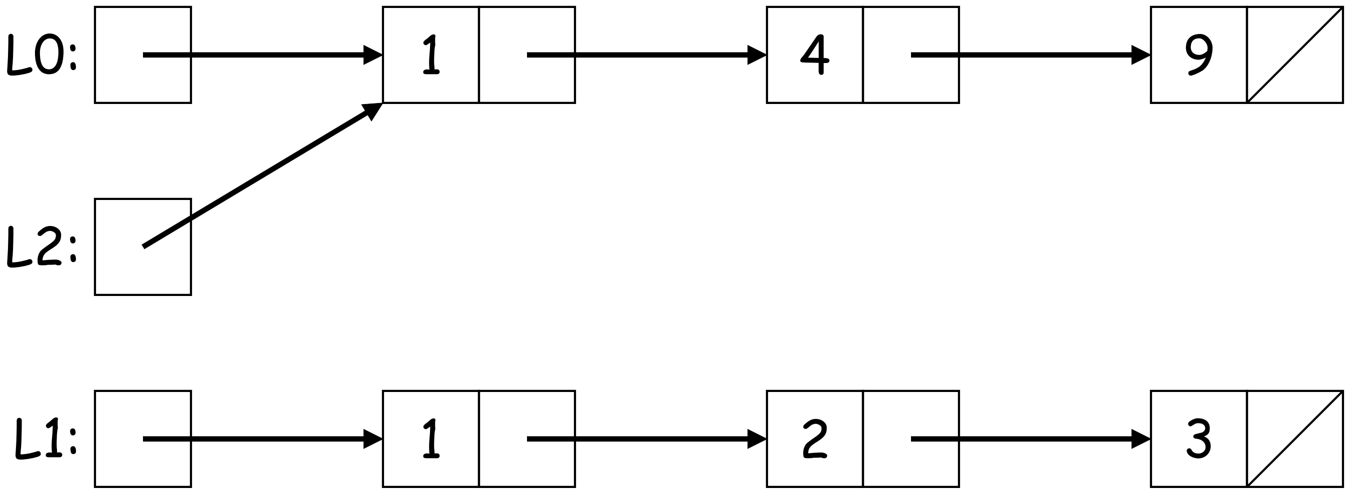
The Picture

- 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)
```



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



Identity

- We distinguish between *identity* of objects:

```
S0 = (1, 2, 3);  S1 = (1, 2, 3)
(S0 is S1)  ==  False
```

- And *equality of contents:*

```
(S0 == S1)  ==  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] == 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)

```
>>> 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
```