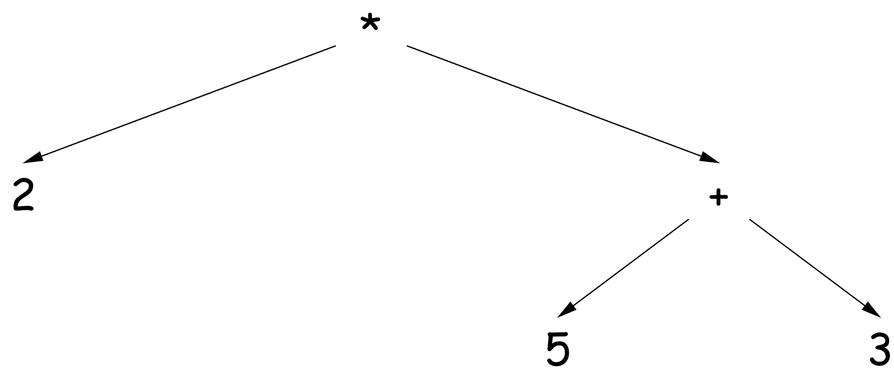
Lecture #12: Immutable and Mutable Data

Listing Leaves

```
def leaf_labels(tree):
    """A list of the labels of all leaves in TREE."""
```

Representing Expressions

- An *expression tree* represents an expression, such as `2 * (5+3)`




```
def eval(expr):
    """The value yielded by the computation represented by
    expression tree EXPR.  Assumes all leaves are numbers
    and all inner-node labels are operators."""
```

Building Recursive Structures

- In Lecture #10, 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 in 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 treat these lists 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*.

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

Still Another Example: Mapping a Tree

- From lecture #11, a tree's recursive structure is:
  - A label and
  - Zero or more children, each a tree.

```
def map_tree(f, T):
    """The tree T with each label, lab, replaced by F(lab)."""

    return ________________________________
    # Hint:  Use the map operation on sequences!
```

Mapping a Tree (II)

- From lecture #11, a tree's recursive structure is:
  - A label and
  - Zero or more children, each a tree.

```
def map_tree(f, T):
    """The tree T with each label, lab, replaced by F(lab)."""

    return make_tree(label(T),
                     map(lambda x: map_tree(f,x), children(T)))

# or

    return make_tree(label(T),
                     [ map_tree(f, x) for x in children(T) ])
```

- What? No base case???!!!

Immutability and Nondestructive Operations

- The functions in this lecture (and in previous ones) did not modify existing list or tree structures.
- 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.
  - Potentially useful in parallel computations.

Mutability and Destructive Operations

- What if we *don't* need the original data?
- Then nondestructive operations have memory costs, possibly time costs as well.
- Suppose we add two more operations to *rlist*:

```
def set_first(r, v):
    """Cause first(R) to be V."""
    R[0] = v
def set_rest(r, V):
    """Cause rest(R) to be V."""
    R[1] = v
```

- To do this, we need to change our implementation of `make_rlist` subtly:

```
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."""
    return [ first, rest ]  ⟸ square brackets
```

- We use a Python list (mutable) instead of a tuple (immutable).

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 ____________________
```

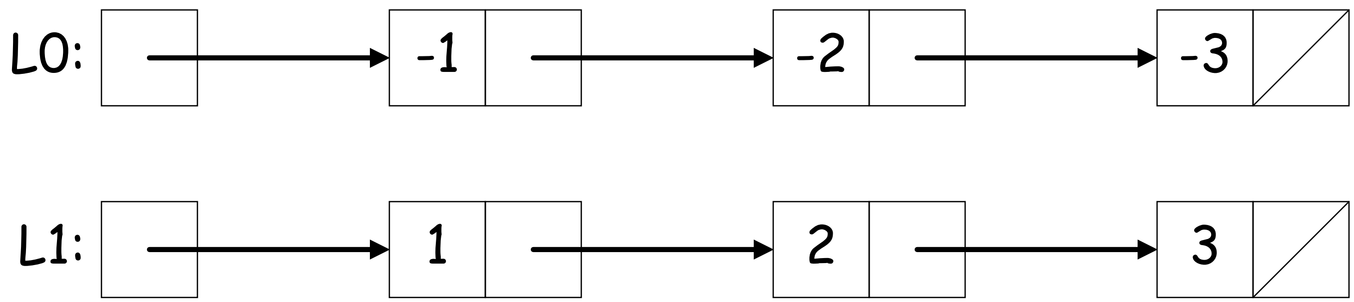
Iterative Version of dmap_rlist (II)

```
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
```

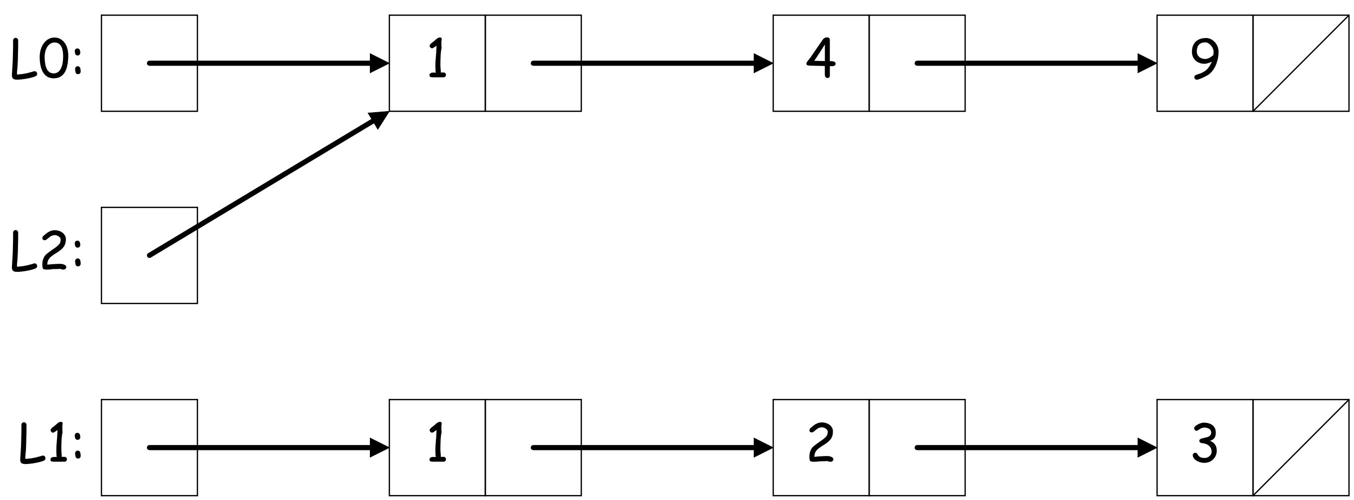
The Picture

- 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

- In a previous lecture, I pointed out the distinction between the *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` not re-assigned, can substitute 3 for `S[0]` anywhere.
- When dealing with mutable structures, identity matters, and we don't have referential transparency.