RLISTS AND THE ENVIRONMENT MODEL 9

COMPUTER SCIENCE 61A

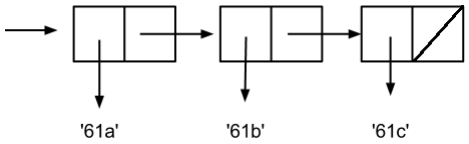
July 17, 2012

1 Recursive Lists (RLists)

Earlier in the course, we worked with the Immutable Recursive List (IRList), which is an abstract data type consisting of a **first** (some value) and a **rest** (another IRList). The end of an IRList was represented by the `empty_irlist`:

```
>>> from irlist import *
>>> c_lst = make_irlist('61a',
                        make_irlist('61b',
                                    make_irlist('61c',
                                                empty_irlist)))
>>> irlist_str(c_lst)
<'61a', '61b', '61c'>
```

For instance, the box-and-pointer diagram of `courses` would be:



This week, we will work with Mutable Recursive Lists (RLists) that are conceptually exactly the same as an IRList. An RList has a first and a rest, but unlike an IRList, you can **modify** the first and rest of an RList.

Here is a shortened class definition of the RList class[1]:

```
class RList(object):
```

[1]The actual class definition of RList is found at: http://inst.eecs.berkeley.edu/ cs61a/su12/lib/rlist.py

```
    the_empty_rlist = EmptyRList()

    @staticmethod
    def empty():
        return RList.the_empty_rlist

    def __init__(self, first, rest=the_empty_rlist):
        self.first = first
        self.rest = rest
    def __len__(self):
        return 1 + len(self.rest)
    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
    def __repr__(self):
        # omitted for brevity
    def __str__(self):
        # omitted for brevity

    class EmptyRList(object):
        def __repr__(self):
            return ``RList.empty()''
        def __str__(self):
            return ``<>''
        def __len__(self):
            return 0
        def __getitem__(self, k):
            raise IndexError(``out of bounds: {0}''.format(k))
```

1.1 Python "magic methods"

Just like `__init__` is the "special" method name that signals the class constructor, there are other method names that Python will handle specially. These include:

`__len__(self)`
: Returns the length of this object. The `len` function will call the object's `__len__` method, if it's defined.

`__getitem__(self, i)`
: Returns the element at index `i`. For instance, when you index into an object, like `myobj[2]`, this is converted into the method call: `myobj.__getitem__(i)`.

`__repr__(self)`
: Returns a string representation of this object with the following rule: if you were to evaluate this string, then it would return the object.

`__str__(self)`
: Returns some string representation of this object. This is much more relaxed than `__repr__`.

There are many other magic methods in Python, which enable your user-defined classes to more-easily integrate with Python's built-in methods and objects[2].

1.2 Practice with RLists

1. (Warmup) Define a procedure `map_rlist` that, given an RList `rlist` and a function `fn`, returns a new RList with all values in `rlist` mapped onto by `fn`. Don't modify the input RList.

   ```
   def map_rlist(rlist, fn):
   ```

2 RLists and Mutation

Because RList instances are **mutable** objects, we can modify them directly, instead of, say, creating a new RList when operating over them. For instance, we can mutate an RList by doing:

```
>>> rlist = RList(3, RList(2, RList(1)))
>>> rlist
<3, 2, 1>
>>> rlist.first = 'meow'
>>> rlist
<'meow', 2, 1>
>>> rlist.rest.first = 'bark'
>>> rlist
<'meow', 'bark', 1>
>>> rlist.rest.rest = RList(RList('hi'), RList(42))
>>> rlist
```

[2] Aside from the magic methods that we explicitly cover, you will not be responsible for knowing all of the magic methods

```
<'meow', 'bark', <<'hi'>, 42>>
```

2.1 More Practice with Rlists

1. (Warmup) Define a procedure `map_rlist` that, given an RList `rlist` and a function `fn`, applies `fn` to every element in `rlist`. Your function should mutate the input `rlist`, and should not create any new RLists.

   ```
   def map_rlist(rlist, fn):
   ```

2. Now, define a procedure `map_rlist_alt` that works just like `map_rlist`, but instead only applies the function to every other element, starting from the first.

   ```
   def map_rlist_alt(rlist, fn):
   ```

3 Environment Model of Evaluation

In this topic, we will formally go over exactly how programs are executed in Python. In fact, it turns out that many other languages use the **Environment Model of Evaluation**.

The main idea of the environment model is: **All evaulation is done with respect to a frame**. For instance, when you type expressions into the interpreter prompt, you are evaluating the expressions with respect to the **global frame**. On the other hand, expressions or statements that are evaluted within a function body is executed with respect to that function's local frame.

A **frame** is a construct that contains two things:

1. All variable bindings that were declared in this frame.
2. An arrow to the parent frame that this frame extends.

The **global** frame is special, in that all built-in functions and constants are defined here, such as `abs`, `True`, and `pow`. By definition, the global frame has no parent frame, and is the parent frame of all other frames.

An **environment** is a series of frames, starting from the current frame to the global frame. You can generate the current environment by following parent pointers until you reach the global frame – the set of frames that you passed through to get to the global frame consists of your current environment.

Here is a concise list of how the Environment Model evaluates an expression:

constants (numbers, strings, etc.)
These are self-evaluating.

variables
Look up the variable name in the current frame. If it doesn't exist in the current frame, look in the parent frame (and the parent frame of that, etc.), and return the first value with the variable name. If you get to the global frame, and still haven't found the variable name, then signal an unbound variable error (called a `NameError` in Python).

function definitions
Create a new function object, and add the function binding to the current frame.

function calls
Evaluate the operands in the current frame. If the function is a primitive (built-in) procedure, then apply the function by "magic". Otherwise, it's a user-defined function: create a new frame F. Bind fn's formal parameters to the evaluated operands' values within F. Make F extend the current frame. Finally, evaluate fn's body with respect to the new frame F.

3.1 Environment Diagrams

In this course, we use **Environment Diagrams** to visually represent the Environment model. An environment diagram consists of two main objects:

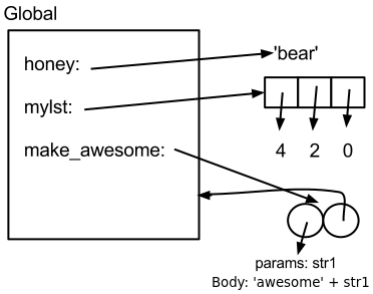
Boxes
These represent your **frames**. Within each box, you'll write out all variable bindings. In addition, each box will have an arrow to its parent frame. You will only create a box when you call a user-defined function. At the very least, you will always have the Global frame present.

Function Bubbles
These represent **user-defined functions**. The left-bubble contains a list of its formal parameters, in addition to the body of the function. The right-bubble contains an arrow that points to the **environment in which the function was created.** You only create function bubbles when a function is created (either via `def` or `lambda`).

In each of your environment diagrams, you'll always start with the **global** frame, but you need not include variable bindings for all built-in functions[3] .

As an example, here's the environment diagram for the following sequence of expressions:

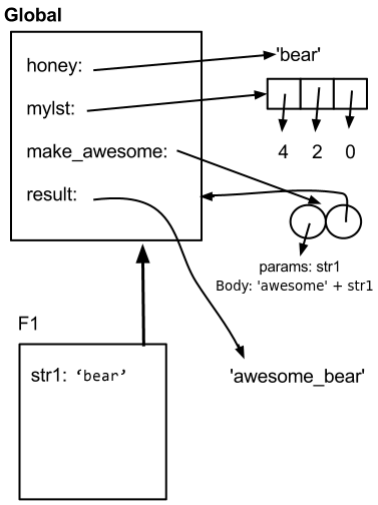
```
>>> honey = 'bear'
>>> mylst = [4, 2, 0]
>>> def make_awesome(str1):
...     return 'awesome_' + str1
```



The interesting thing of note is the function bubble - this object represents the `make_awesome` function. In particular, note that the right-bubble points to the global frame - this is because `make_awesome` was defined in the global frame!

Notice that no new frames have been created yet (we haven't called any user-defined functions yet!). But when we do call `make_awesome`, the following new frame is created:

```
>>> result = make_awesome(honey)
```



[3] At times, you might find it helpful to include some built-in functions in your diagrams – see the environment diagram exercise 2.f for an example. If you do, instead of creating a lambda bubble for the built-in functions, use some distinguishing object that signals that this is a built-in function object, not a user-defined function object.

3.2 Practice with Environment Diagrams

1. (Warmup) Draw the environment diagram of the following expressions:

```
a = lambda x: x*2 + 1
def b(x):
    return x * y
y = 3
b(y)       # What is the return value?
```

DRAW DIAGRAM HERE

```
a = lambda x: x*2 + 1
b = lambda x: x * y
y = 3
def c(x):
    y = a(x)
    return b(x) + a(x+y)
c(y)     # What is the return value?
```

DRAW DIAGRAM HERE

2. Draw the environment diagram at the indicated locations. If an error occurs, write ERROR:

```
def square(x):
    return x * x
def call_it(fn, n):
    return fn(n)
n = 22
call_it(square, n-2)     # What is the return value?
```

DRAW DIAGRAM HERE

```
def make_adder(x):
    return lambda y: x + y
add_4 = make_adder(4)
add_4(5)    # What is the return value?
```

DRAW DIAGRAM HERE

```
mylst = [1, 2, 'yay']
def modify_it_maybe(lst):
    lst = ['this', 'is', 'crazy']
    return lst
result = modify_it_maybe(mylst)
result[2] = 'awesome'
mylst[0] = 'wat'
```

DRAW DIAGRAM HERE

```
val = 3
square = lambda x: x * x
def foo(fn, val):
    def doit():
```

```
            return fn(val)
        return doit
f = foo(square, 4)
f()     # What is the return value?
```

DRAW DIAGRAM HERE

```
from operator import add, sub
def f(add, sub):
    return add((lambda sub: sub(3, 5))(sub), 10)
f(sub, add)     # What is the return value?
```

DRAW DIAGRAM HERE

```
def hmm(n):
    return lambda x: x + y + n

def uhh(y):
    hmm_y = hmm(y)
    return hmm_y(2)

uhh(42)     # What is the return value?
```

DRAW DIAGRAM HERE