CALCULATOR 10

COMPUTER SCIENCE 61A

July 24, 2014

Remember homework 6? Let's take a second look at the **Calculator** language, a subset of a language we'll be learning later called Scheme. In today's discussion, we'll be looking at implementing Calculator using regular Python.

1 Calculator

Here is a reminder of how our Calculator language looks so far. Right now it can handle the four basic arithmetic operations which can be nested and take varying numbers of arguments. Here's a couple examples of Calculator in action:

```
> (+ 2 2)
4
> (- 5)
-5
> (* (+ 1 2) (+ 2 3))
15
```

The goal is to write an interpreter for this Calculator language. The job of an interpreter is, given an expression, evaluate its meaning. So let's talk about expressions.

1.1 Representing Expressions

There are two kinds of expressions. A **call expression** is a linked list - where the first element is the operator, and each subsequent element is an operand. A **primitive expression** is an operator symbol or number. When we type a line at the Calculator prompt and hit enter, we've just sent an expression to the interpreter.

To represent Scheme lists in Python, we used `Pair` objects. The class definition is below:

```
class nil:
    """The empty list"""

    def __len__(self):
        return 0

    def map(self, fn):
        return self

    def to_py_list(self):
        return []

nil = nil() #nil now refers to a single instance of nil class

class Pair:

    def __init__(self, first, second=nil):
        self.first = first
        self.second = second

    def __len__(self):
        n, second = 1, self.second
        while isinstance(second, Pair):
            n += 1
            second = second.second
        if second is not nil:
            raise TypeError("length attempted on improper list")
        return n

     def __getitem__(self, k):
        if k < 0:
            raise IndexError("negative index into list")
        j, y = 0, self
        while j < k:
            if y.second is nil:
                raise IndexError("list index out of bounds")
            elif not isinstance(y.second, Pair):
                raise TypeError("ill-formed list")
            j, y = j + 1, y.second
        return y.first
    def map(self, fn):
        """Returns a Scheme list after mapping Python function
        fn over self."""
        mapped = fn(self.first)
        if self.second is nil or isinstance(self.second, Pair):
            return Pair(mapped, self.second.map(fn))
        else:
            raise TypeError("ill-formed list")

   def to_py_list(self):
        """Returns a Python list containing the elements of this
        Scheme list."""
        y, result = self, [ ]
        while y is not nil:
            result += [y.first]
            if not isinstance(y.second, Pair) and y.second is not nil:
                raise TypeError("ill-formed list")
            y = y.second
        return result
```

1.2 Questions

1. Translate the following Python representation of Calculator expressions into the proper Scheme-syntax:

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))

>>> Pair('+', Pair('1', Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

Solution:

```
> (+ 1 2 3 4)
> (+ 1 (* 2 3))
```

2. Translate the following Calculator expressions into calls to the `Pair` constructor.

```
> (+ 1 2 (- 3 4))

> (+ 1 (* 2 3) 4)
```

Solution:

```
>>> Pair('+', Pair(1, Pair(2, Pair(
          Pair('-', Pair(3, Pair(4, nil))), nil))))
>>> Pair('+', Pair(1, Pair(Pair('*', Pair(
          2, Pair(3,nil))), Pair(4,nil))))
```

1.3 Evaluation

So what is evaluation? Evaluation discovers the form of an expression and executes a corresponding evaluation rule.

Primitive expressions are evaluated directly. Call expressions are evaluated recursively:

1. Evaluate each operand expression,
2. Collect their values as a list of arguments, and
3. **Apply** the named operator to the argument list.

Here's calc_eval:

```
def calc_eval(exp):
    if not isinstance(exp, Pair): # expression is primitive
        return exp
    else:
        operator, operands = exp.first, exp.second
        args =  operands.map(calc_eval).to_py_list()
    return calc_apply(operator, args)
```

As you can see, all we've done is follow the rules of evaluation outlined above. If the expression is primitive (i.e. not a Scheme list), simply return it. Otherwise, evaluate the operands and apply the operator to the evaluated operands.

How do we apply the operator? We'll use calc_apply, with dispatching on the operator name:

```
def calc_apply(operator, args):
    if operator == '+':
        return sum(args)
    elif operator == '-':
        if len(args) == 1:
            return -args[0]
        else:
            return args[0] - sum(args[1:])
    elif operator == '*':
        return reduce(mul, args, 1)
```

Depending on what the operator is, we can match it to a corresponding Python call. Each conditional clause above handles the application of one operator.

Something very important to keep in mind: `calc_eval` deals with **expressions**, `calc_apply` deals with **values**.

1.4 Questions

1. Suppose we typed each of the following expressions into the Calculator interpreter. How many calls to `calc_eval` would they each generate? How many calls to `calc_apply`?

```
> (+ 2 4 6 8)                > (+ 2 (* 4 (- 6 8)))
```

Solution:

```
> (+ 2 4 6 8)
5 calls to eval. 1 call to apply.
> (+ 2 (* 4 (- 6 8)))
7 calls to eval. 3 calls to apply.
```

2. The - operator will fail if given no arguments. Add error handling to raise an exception when this situation is encountered (the type of exception is unimportant).

Solution:

```
...
    if operator == '-':
        if len(args) == 0:
          raise TypeError('need at least one arg')
...
```

3. We also want to be able to perform division, as in `(/ 4 2)`. Supplement the existing code to handle this. If division by 0 is attempted, or if there are no arguments supplied, you should raise an exception (the type of exception is unimportant).

Solution:

```
...
    if operator == '/':
        if len(args) == 0:
          raise TypeError('need at least one arg')
        elif len(args) ==  1:
          return 1/args[0]
        elif 0 in args[1:]:
          raise ZeroDivisionError
        else:
          return reduce(truediv, args[1:], args[0])
...
```

4. Alyssa P. Hacker and Ben Bitdiddle are also tasked with implementing the `and` operator, as in `(and (= 1 2) (< 3 4))`. Ben says this is easy: they just have to follow the same process as in implementing `*` and `/`. Alyssa is not so sure. Who's right?

Solution: Alyssa. We can't handle `and` in the apply step since `and` is a special form: it is short-circuited. We need to create a special case for it in `calc_eval`.

5. Now that you've had a chance to think about it, you decide to try implementing `and` yourself. You may assume the conditional operators (e.g. <, >, =, etc) have already been implemented for you.

Solution:

```
def calc_eval(exp):
    if not isinstance(exp, Pair):
        return exp
    elif exp.first == 'and':
        return eval_and(exp.second)
    else:
        ...

def eval_and(operands):
    cur = operands
    while cur is not nil:
        if not calc_eval(cur.first):
            return False
        cur = cur.second
    return True
```

1.5 Bonus Questions

6. Implement the `expt` operator such that the following works:

```
> (expt 2 3)
8
> (expt 3 4)
81
```

Solution:

```
...
    if operator == 'expt':
        return args[0] ** args[1]
...
```

7. Implement `quote`. A `quote` expression simply returns its argument without evaluating it.

```
> (quote (2 (3 4) 5))
(2 (3 4) 5)
> (quote (+ 3 4))
(+ 3 4)
```

Solution:

```
def calc_eval(exp):
    ...
    elif exp.first == "quote":
        return exp.second.first
    ...
```

8. Implement the `list` function. `list` evaluates all its arguments and returns a list of their values.

```
> (list (+ 3 4) 5 (* 2 3))
(7 5 6)
```

```
> (list (+ 1 2) (quote (3 4)) 5)
(3 (3 4) 5)
```

Solution:

```
def calc_apply(operator, args):
    ...
    elif operator == "list":
        return list_to_pairs(args)
    ...
def list_to_pairs(lst):
    if len(lst)==0:
        return nil
    return Pair(lst[0], list_to_pairs(lst[1:]))
```

9. Now that we can create lists, let's modify the + operator so that it can add lists together elementwise. You can assume that the lists are the same length and contain only numbers.

```
> (+ (quote (7 4 3 9)) (quote 1 1 1 1) (quote (5 1 5 1)))
(13 6 9 2)
> (+ (quote (1 2 3 4)) (list (+ 2 2) 3 (- 4 2) 1))
(5 5 5 5)
```

Solution:

```
def calc_apply(operator, args):
    ...
    if operator == '+':
        if type(args[0])==Pair:
            return reduce(lambda x, y: add_pairs(x,y), args[1:], args[0
        return sum(args)
    ...
def add_pairs(x, y):
    if (x is nil) or (y is nil):
        return nil
    return Pair(x.first + y.first, add_pairs(x.second, y.second))
```

2 Trees as Classes

2.1 Trees

Before, we represented trees using functions. Now that we've learned OOP, let's see `Trees` in a whole new way (but not really)! A tree has a `datum` and a list of `children` as before.

```
class Tree:
    def __init__(self, datum, *args):
        self.datum = datum
        self.children = list(args)
```

1. Implement the method `size` which returns the size of a tree.

```
    def size(self):
        """
        >>> t = Tree(9, Tree(1, Tree(1), Tree(9)), Tree(1, Tree(1)))
        >>> t.size()
        6
        """
```

Solution:

```
        s = 1
        for tree in self.children:
            s += tree.size()
        return s
```

2. Implement the `height` method. The height of a tree is the length of the longest path to a leaf.

```
    def height(self):
        """
        >>> t = Tree(1, Tree(2, Tree(4), Tree(5, (Tree(7)))), \
            Tree(3, Tree(6)))
        >>> t.children[1].height() # Tree whose datum is 3
        1
        >>> t.height() # longest path
        3
        """
```

Solution:

```
        if self.children == []:
            return 0
        return max([1+child.height() for child in self.children])
```

3. The best part about classes are their mutability. Let's make a method `replace` that replaces all data of some value in a tree with a new value.

```
def replace(self, old, new):
    """
    >>> t = Tree(1, Tree(2, Tree(3), Tree(2)), Tree(3, Tree(2)))
    >>> t.replace(3, 1)
    >>> t.pretty_print()
    1
    |___2
    |   |___1
    |   |___2
    |___1
        |___2
    """
```

Solution:

```
        if self.datum == old:
            self.datum = new
        for child in self.children:
            child.replace(old, new)
```

4. Implement a method `add_tree` that adds a child to the end of the tree.

```
def add_tree(self, tree):
    """
    >>> t = Tree(1, Tree(2))
    >>> t.add_tree(Tree(3, Tree(4)))
    >>> t.pretty_print()
    1
    |___2
    |___3
        |___4
    """
```

Solution:

```
        self.children.append(tree)
```

5. Implement a `get_tree` method that returns the node in the tree whose `datum` is equal to `val`, or `None` if no such node exists. You may assume that there are no duplicate elements in the tree.

```
    def get_tree(self, val):
        """
        >>> t = Tree(1, Tree(2, Tree(4), Tree(5)), Tree(3, Tree(6)))
        >>> t.get_tree('foo')
        >>> t.get_tree(3).pretty_print()
        3
        |__6
        """
```

Solution:

```
        if self.datum == val:
            return self
        for child in self.children:
            d = child.get_tree(val)
            if d:
                return d
```

2.2 Binary Trees

A Binary Tree is a tree in which every node has at most two children, one to the `left` and one to the `right`.

```
class BinaryTree(object):
    def __init__(self, datum, left=None, right=None):
        self.datum, self.left, self.right = datum, left, right
```

1. Implement the method `size` to find the size of the tree.

```
    def size(self):
        """ Returns the size of a tree.
        >>> t = BinaryTree(7, BinaryTree(3,\
        BinaryTree(4), BinaryTree(6)))
        >>> t.size()
        4
        >>> BinaryTree(3, None, BinaryTree(5)).size()
        2
        """
```

Solution:

```
        result = 1
        if self.left:
            result += self.left.size()
        if self.right:
            result += self.right.size()
        return result
```

2. A `Binary Search Tree` (BST) is a Binary Tree in which all datum values in the `left` child of a tree are less than its `datum`, and all datum values to the right are greater than its `datum`. Make a method `is_bst` that determines whether a Binary Tree is also a BST.

```
    def is_bst(self, low=float('-inf'), high=float('inf')):
        """ Makes sure a tree is a bst.
        >>> t = BinaryTree(5, BinaryTree(3), BinaryTree(6))
        >>> t.pretty_print()
          5
         / \
        3   6
        >>> t.is_bst()
        True
        >>> s = BinaryTree(7, BinaryTree(4,\
          BinaryTree(3), BinaryTree(8)), None)
        >>> s.pretty_print()
            7
           /
          4
         / \
        3   8
        >>> s.is_bst() # 8 is bigger than 7 though it is on the left
        False
        """
```

Solution:

```
        if self.datum < low or self.datum > high:
            return False
        if self.left and not self.left.is_bst(low, self.datum):
            return False
        if self.right and not self.right.is_bst(self.datum, high):
            return False
        return True
```

2.3 Binary Search Trees

You can also make a BST class that inherits from a Binary Tree.

```
class BST(BinaryTree):
    def __init__(self, datum, left=None, right=None):
        BinaryTree.__init__(self, datum, left, right)
        assert self.is_bst(), "Breaking Invariants"
```

As a fundamental quirk of class objects is their mutability, it's important to create a way to add to our BST while maintaining it's properties of being sorted.

1. Implement a method `insert` which will insert an element into the tree. Make sure you preserve the BST invariant (things to the left are smaller, things to the right are larger). If the element is already present, don't add it to the tree.

```
def insert(self, elem):
    """
    >>> t = BST(15, BST(7))
    >>> t.insert(12)
    >>> t.insert(28)
    >>> t.insert(7)
    >>> t.pretty_print()
      15
     / \
    7   28
     \
      12
    """
```

Solution:

```
        if elem == self.datum:
            return # do nothing
        elif elem < self.datum:
            if self.left == None:
                self.left = BST(elem)
            else:
                self.left.insert(elem)
        else:
            if self.right == None:
                self.right = BST(elem)
            else:
                self.right.insert(elem)
```