CALCULATOR 10

COMPUTER SCIENCE 61A

November 6, 2014

We are beginning to dive into the realm of interpreting computer programs – that is, writing programs that understand other programs. In order to do so, we'll have to examine programming languages in-depth. The *Calculator* language, a subset of Scheme, will be the first of these examples.

In today's discussion, we'll be implementing Calculator using regular Python.

1 Calculator

The Calculator language is a Scheme-syntax language that includes only the four basic arithmetic operations: $+$, $-$, $*$, and /. These operations can be nested and can take varying numbers of arguments. Here's a few examples of Calculator in action:

```
> (+ 2 2)
4

> (- 5)
-5

> (* (+ 1 2) (+ 2 3))
15
```

Our goal now is to write an interpreter for this Calculator language. The job of an interpreter is to evaluate expressions. So, let's talk about expressions.

1.1 Representing Expressions

A Calculator expression is just like a Scheme list. To represent Scheme lists in Python, we use `Pair` objects. For example, the list `(+ 1 2)` is represented as `Pair('+', Pair(1, Pair(2, nil)))`. The `Pair` class is similar to the Scheme procedure `cons`, which would represent the same list as `(cons '+ (cons 1 (cons 2 nil)))`.

`Pair` is very similar to `Link`, the class we developed for representing linked lists. In addition to `Pair` objects, we include a `nil` object to represent the empty list. Both `Pair` instances and `nil` have methods:

1. `__len__`, which returns the length of the list.
2. `__getitem__`, which allows indexing into the pair.
3. `apply_to_all`, which applies a function, `fn`, to all of the elements in the list.
4. `to_py_list`, which returns a Python list with the same elements.

Here's an implementation of what we described:

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

    def __len__(self):
        return 0

    def apply_to_all(self, fn):
        return self

nil = nil() # this hides the nil class *forever*

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

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

    def __getitem__(self, k):
        if k == 0:
            return self.first
```

```
        if k < 0:
            raise IndexError("negative index into list")
        elif self.second is nil:
            raise IndexError("list index out of bounds")
        elif not isinstance(self.second, Pair):
            raise TypeError("ill-formed list")
        return self.second[k-1]

    # Note: this method was called "map" in lecture
    def apply_to_all(self, fn):
        """Returns a Scheme list after applying Python function
        fn over self."""
        applied = fn(self.first)
        if self.second is nil or isinstance(self.second, Pair):
            return Pair(applied, self.second.apply_to_all(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.append(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)))
```

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

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

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

1.3 Evaluation

Evaluation discovers the form of an expression and executes a corresponding evaluation rule.

We'll go over two such expressions now:

1. *Primitive* expressions are evaluated directly. e.g. "1" just evaluates to itself.
2. *Call* expressions are evaluated in the same way you've been doing them by hand all semester:
   (1) **Evaluate** the operator.
   (2) **Evaluate** the operands from left to right.
   (3) **Apply** the operator to the operands.

Here's `calc_eval`:

```
def calc_eval(exp):
    if not isinstance(exp, Pair): # primitive expression
        return exp
    else: # call expression

        # Step 1: evaluate the operator.
        operator = exp.first

        # Step 2: evaluate the operands.
        operands = exp.second
        args =  operands.apply_to_all(calc_eval).to_py_list()

    # Step 3: apply the operator to the operands.
    return calc_apply(operator, args)
```

How do we apply the operator? We'll dispatch on the operator name with `calc_apply`:

```
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** (in Calculator), `calc_apply` deals with **values** (which are in Python).

1.4 Exceptions

Recall that **exceptions** are used to signify when something goes wrong in your program. For interpreters, they're often used to categorize a case when the user inputs something that doesn't make sense (just try typing in `Hi Soumya` in your Python interpreter and see what happens!)

There are two major things that you do with exceptions: `raise` and **handle** them.

Generally, to raise an exception you use the statement `raise <expression>`.

To handle an exception, you use a `try-except` block. The syntax is as follows:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

You can have multiple `except` suites for different types of exceptions that might occur in the try suite.

1.5 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)))
```

2. The – operator will fail if given no arguments. Add error handling to raise a `TypeError` when this situation is encountered (the error message is unimportant).

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, raise a `ZeroDivisionError`. If there are less than 2 arguments supplied, raise a `TypeError` (the error message is unimportant).

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?

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.

1.6 Extra Practice

1. 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)
```

2. Implement the list operator. A list expression 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)
```

3. Now that we can create Scheme-style lists in calculator, lets 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 6 2 6 2))
(13 6 9 2)
> (+ (quote (1 2 3 4)) (list (+ 2 2) 3 (/ 4 2) 1))
(5 5 5.0 5)
```