1 Calculator

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, was the first of these examples. In today's discussion, we'll be extending Calculator with variables and user-defined functions.

The Calculator language is a Scheme-syntax language that currently includes only the four basic arithmetic operations: $+$, $-$, $*$, and $/$. These operations can be nested and can take varying numbers of arguments. A few examples of calculator in action are given on the right.

Our goal now is to write an interpreter for this language, and extend its functionality to variables and user-defined functions. The job of an interpreter is to evaluate expressions. So, let's talk about 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 the same as the Scheme procedure `cons`, which would represent the same list as `(cons '+ (cons 1 (cons 2 nil)))`.

```
calc> (+ 2 2)
4

calc> (- 5)
-5

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

`Pair` is very similar to `Link`, the class we developed for representing linked lists, except that the second attribute doesn't have to be a linked list. In addition to `Pair` objects, we include a `nil` object to represent the empty list. `Pair` instances have methods:

1. `__len__`, which returns the length of the list.
2. `__getitem__`, which allows indexing into the pair.
3. `map`, which applies a function, `fn`, to all of the elements in the list.

`nil` has the methods `__len__`, `__getitem__`, and `map`. Here's an implementation of what we described:

```
class nil:
    """Represents the special empty pair nil in Scheme."""
    def __repr__(self):
        return 'nil'
    def __len__(self):
        return 0
    def __getitem__(self, i):
        raise IndexError('Index out of range')
    def map(self, fn):
        return nil

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

```
class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return 'Pair({}, {})'.format(self.first, self.second)
    def __len__(self):
        return 1 + len(self.second)
    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.second[i-1]
    def map(self, fn):
        return Pair(fn(self.first), self.second.map(fn))
```

Questions

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

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

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

1.2 Translate the following Python representations 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 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. For example, the numbers 3.14 and 165 just evaluate to themselves, and the string "+" evaluates to the `calc_add` function.
2. *Call* expressions are evaluated in the same way you've been doing them 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):
    """Evaluates a Calculator expression represented as a Pair."""
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                          list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:  # Primitive expression
        return exp
```

And here's `calc_apply`:

```
def calc_apply(op, args):
    """Applies an operator to a Pair of arguments."""
    return op(*args)
```

Questions

2.1 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)))
```

2.2 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?

2.3 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.

```
def calc_eval(exp):
```

```
def eval_and(operands):
```

3 Tail-Call Optimization

3.1 Write a tail recursive function that returns the nth fibonacci number. We define $\text{fib}(0) = 0$ and $\text{fib}(1) = 1$.

```
(define (fib n)
```

3.2 Write a tail recursive function, `reverse`, that takes in a Scheme list and returns a reversed copy.

```
(define (reverse lst)
```

3.3 Write a tail recursive function, `insert`, that takes in a number and a sorted list. The function returns a sorted copy with the number inserted in the correct position.

```
(define (insert n lst)
```

3.4 Write a tail recursive function, `append`, that takes in two lists and appends them. Make sure that your function is $\Theta(n)$ and tail-recursive.

```
(define (append a b)
```