We are beginning to dive into the realm of interpreting computer programs - that is, writing programs that understand 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 looking at implementing Calculator using regular Python. We’ll also take a look at Exceptions, a mechanism for handling unexpected execution - quite common when handling user input.

1 Calculator

For now, our Calculator language will be a Scheme-syntax language that can handle the four basic arithmetic operations. These operations can be nested and can take varying numbers of arguments. Here’s a couple 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, 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 Scheme list - 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’ll be using Pair objects. This is very similar to the Rlists we’ve been working with. The class definition is below:

class nil(object):
    """The empty list""

    def __len__(self):
        return 0

    def map(self, fn):
        return self

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

class Pair(object):

    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:
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) or 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:**

```python
>>> Pair('+', Pair(1, Pair(2, Pair(
P  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`:

```python
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:

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

CS61A Fall 2013: John DeNero, with
Soumya Basu, Jeff Chang, Brian Hou, Andrew Huang, Robert Huang, Michelle Hwang, Richard Hwang, Joy Jeng, Keegan Mann, Stephen Martinis, Bryan Mau, Mark Miyashita, Allen Nguyen, Julia Oh, Vaishaal Shankar, Steven Tang, Sharad Vikram, Albert Wu, Chenyang Yuan
```python
else:
    return sum(args[0], [-args for args in 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?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Calls to calc_eval</th>
<th>Calls to calc_apply</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 2 4 6 8)</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>(+ 2 (* 4 (- 6 8)))</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

   **Solution:**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Calls to Calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 2 4 6 8)</td>
<td>5 calls to eval. 1 call to apply.</td>
</tr>
<tr>
<td>(+ 2 (* 4 (- 6 8)))</td>
<td>7 calls to eval. 3 calls to apply.</td>
</tr>
</tbody>
</table>

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:**

   ```python
   ...
   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 less than 2 arguments supplied, you should raise an exception (the type of exception is unimportant).
4. Alyssa P. Hacker and Ben Bitdiddle are also tasked with implementing the \texttt{and} operator, as in \((\texttt{and} \ (1 \ 2) \ (< \ 3 \ 4))\). Ben says this is easy: they just have to follow the same process as in implementing \texttt{*} and \texttt{/}. Alyssa is not so sure. Who’s right?

\begin{quote}
\textbf{Solution:} Alyssa. We can’t handle \texttt{and} in the apply step since \texttt{and} is a special form: it is short-circuited. We need to create a special case for it in \texttt{calc_eval}.
\end{quote}
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:
```python
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:
```python
... if operator == 'expt':
    return args[0] ** args[1]
...
```

7. Implement quote. A quote expression simply returns its argument without evaluating
it.
8. Implement the list operator. A list expression evaluates all its arguments and returns a list of their values.

\[
\begin{align*}
\text{> } & (\text{list } (+ \ 3 \ 4) \ 5 \ (* \ 2 \ 3)) \\
& (7 \ 5 \ 6) \\
\text{> } & (\text{list } (+ \ 1 \ 2) \ (\text{quote } (3 \ 4)) \ 5) \\
& (3 \ (3 \ 4) \ 5)
\end{align*}
\]

Solution:

```python
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 scheme-style lists in calculator, 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.

\[
\begin{align*}
\text{> } & (+ \ (\text{quote } (7 \ 4 \ 3 \ 9)) \ (\text{quote } 6 \ 2 \ 6 \ 2)) \\
& (13 \ 6 \ 9 \ 2) \\
\text{> } & (+ \ (\text{quote } (1 \ 2 \ 3 \ 4)) \ (\text{list } (+ \ 2 \ 2) \ 3 \ (/ \ 4 \ 2) \ 1)) \\
& (5 \ 5 \ 5.0 \ 5)
\end{align*}
\]
Solution:

```python
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))
```