Discussion 11: Interpreters

This is an online worksheet that you can work on during discussions and tutorials. Your work is not graded and you do not need to submit anything.

Discussion 11 Vitamin

To encourage everyone to watch or attend discussion orientation, we have created small discussion vitamins. If you complete 5 of the 6 vitamins, you can earn one point of extra credit added to your final grade in the course. Please answer all of the questions in this form (https://links.cs61a.org/vitamin-for-disc11) by **Thursday at 11:59 PM**.

Calculator

An interpreter is a program that understands other programs. Today, we will explore how to build an interpreter for Calculator, a simple language that uses a subset of Scheme syntax.

The Calculator language includes only the four basic arithmetic operations: `+`, `-`, `*`, and `/`. These operations can be nested and can take any numbers of arguments. A few examples of calculator expressions and their corresponding values are shown below.

```
calc> (+ 2 2)
4

calc> (- 5)
-5

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

The reader component of an interpreter parses input strings and represents them as data structures in the implementing language. In this case, we need to represent Calculator expressions as Python objects. To represent numbers, we can just use Python numbers. To represent the names of the arithmetic procedures, we can use Python strings (e.g. `'+'`).

Call expressions are a bit more complicated. Like Scheme call expressions, Calculator call expressions look just like Scheme lists. For example, to construct the expression `(+ 2 3)` in Scheme, we would do the following:

```
scm> (cons '+ (cons 2 (cons 3 nil)))
(+ 2 3)
```

To represent Scheme lists in Python, we will use the `Pair` class. A `Pair` instance holds exactly two elements. Accordingly, the `Pair` constructor takes in two arguments, and to make a list we must nest calls to the constructor and pass in `nil` as the second element of the last pair. Note that in our implementation, `nil` is bound to a special user-defined object that represents an empty list, whereas `nil` in Scheme is actually an empty list.

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

Each `Pair` instance has two instance attributes: `first` and `rest`, which are bound to the first and second elements of the pair respectively.

```
>>> p = Pair('+', Pair(2, Pair(3, nil)))
>>> p.first
'+'
>>> p.rest
Pair(2, Pair(3, nil))
>>> p.rest.first
2
```

`Pair` is very similar to `Link`, the class we developed for representing linked lists -- they have the same attribute names `first` and `rest` and are represented very similarly. Here's an implementation of what we described:

```
class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, rest):
        self.first = first
        if not scheme_valid_cdrp(rest):
            raise SchemeError("cdr can only be a pair, nil, or a promise but was {}".format(r
        self.rest = rest

    def map(self, fn):
        """Maps fn to every element in a list, returning a new
        Pair.

        >>> Pair(1, Pair(2, Pair(3, nil))).map(lambda x: x * x)
        Pair(1, Pair(4, Pair(9, nil)))
        """
        assert isinstance(self.rest, Pair) or self.rest is nil, \
            "rest element in pair must be another pair or nil"
        return Pair(fn(self.first), self.rest.map(fn))

    def __repr__(self):
        return 'Pair({}, {})'.format(self.first, self.rest)
```

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

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

Questions

Q1: From Pair to Calculator

Write out the Calculator expression with proper syntax that corresponds to the following `Pair` constructor calls. Also, draw out a box and pointer diagram corresponding to each input.

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

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

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

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

Q2: Using Pair

Answer the following questions about a `Pair` instance representing the Calculator expression `(+ (- 2 4) 6 8)`.

Write out the Python expression that returns a `Pair` representing the given expression:

Draw a box and pointer diagram corresponding to the Pair:

.
.
.
.

What is the operator of the call expression?

If the `Pair` you constructed in the previous part was bound to the name `p`, how would you retrieve the operator?

What are the operands of the call expression?

If the `Pair` you constructed was bound to the name `p`, how would you retrieve a list containing all of the operands?

How would you retrieve only the first operand?

Evaluation

The evaluation component of an interpreter determines the type of an expression and executes corresponding evaluation rules.

Here are the evaluation rules for the three types of Calculator expressions:

1. **Numbers** are self-evaluating, and can be thought of as primitives. For example, the numbers 3.14 and 165 just evaluate to themselves.
2. **Names** are looked up in the `OPERATORS` dictionary. In this dictionary, each name (e.g. `'+'`) is bound to a corresponding function in Python that does the appropriate operation on a list of numbers (e.g. `sum`).
3. **Call expressions** are evaluated the same way you've been doing them all semester:
    1. **Evaluate** the operator, which evaluates to a function.
    2. **Evaluate** the operands from left to right.
    3. **Apply** the function to the value of the operands.

The function `calc_eval` takes in a Calculator expression represented in Python and implements each of these rules:

```
def calc_eval(exp):
    """Evaluates a Calculator expression represented as a Pair."""
    if isinstance(exp, Pair): # Call expressions
        fn = calc_eval(exp.first)
        args = list(exp.rest.map(calc_eval))
        return calc_apply(fn, args)
    elif exp in OPERATORS:    # Names
        return OPERATORS[exp]
    else:                     # Numbers
        return exp
```

`calc_eval` is recursive! In order to evaluate call expressions, we must call `calc_eval` on the operator and each of the operands.

The `apply` step in the Calculator language is straight-forward since we only have primitive procedures. This step will be more complex in the Scheme project since the procedures may include user-defined procedures.

Given the Python function that implements the appropriate Calculator operation and a Python list of numbers, the `calc_apply` function simply calls the function on the arguments, and regular Python evaluation rules take place.

```
def calc_apply(fn, args):
    """Applies a Calculator operation to a list of numbers."""
    return fn(args)
```

Questions

Q3: Counting Eval and Apply

How many calls to `calc_eval` and `calc_apply` would it take to evaluate each of the following Calculator expressions?

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

20

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

-6

Q4: New Operators

Suppose we want to add handling for comparison operators `>`, `<`, and `=` as well as `and` expressions to our Calculator interpreter. These should work the same way they do in Scheme.

```
calc> (and (= 1 1) 3)
3
calc> (and (+ 1 0) (< 1 0) (/ 1 0))
#f
```

i. Are we able to handle expressions containing the comparison operators (such as `<`, `>`, or `=`) with the existing implementation of `calc_eval`? Why or why not?

ii. Are we able to handle `and` expressions with the existing implementation of `calc_eval`? Why or why not?

> **Hint:** Think about the rules of evaluation we've implemented in `calc_eval`. Is anything different about `and`?

iii. Now, complete the implementation below to handle `and` expressions. You may assume the conditional operators (e.g. `<`, `>`, `=`, etc) have already been implemented for you.

```
    4             return eval_and(exp.rest)
    5         else:                   # Call expressions
    6             return calc_apply(calc_eval(exp.first), list(exp.rest.map(calc_eval)))
```

```
    elif exp in OPERATORS:      # Names
        return OPERATORS[exp]
    else:                       # Numbers
        return exp

def eval_and(operands):
    "*** YOUR CODE HERE ***"
    ans = operands[0]
    for operand in operands[:]:
        ans = ans and operand
    return ans

```

Interpreters Tutorial Discussions

Q5: (Tutorial) Interpreters Review

Discuss the follow questions with your tutorial group - they will be helpful for your understanding of the Scheme project! If you wish to take notes, we recommend you take notes on a separate document so it won't accidentally get erased.

What are the four parts of an interpreter (Hint: what does REPL stand for)? What does each part do? What parts did you work on implementing in the discussion?

For the Calculator interpreter implemented in discussion, for the following executed code, what would be the input into the "Read" portion of the interpreter?

```
calc> (+ 2 3)
5
```

What would be the output of the "Read" portion for the same code?

How does the evaluate stage work in Calculator? How do we know if an input into `calc_eval` is a call expression?

Scheme Lists

Q6: (Tutorial) Replicate

Write a function that takes an element `x` and a non-negative integer `n`, and returns a list with `x` repeated `n` times.

> **Tip:** If you aren't sure where to start, try writing the corresponding recursive function for Linked Lists in Python first!

```
(define (replicate x n)
  (cond
    ((= n 0) '())
    (else    (cons x (replicate x (- n 1))))))

; ;; Tests
(replicate 5 3)

; expect (5 5 5)
```

```
scm> (replicate 5 3)
(5 5 5)
```

Q7: (Tutorial) Run Length Encoding

A **run-length encoding** is a method of compressing a sequence of letters. The list `(a a a b a a a a)` can be compressed to `((a 3) (b 1) (a 4))`, where the compressed version of the sequence keeps track of how many letters appear consecutively.
Write a function that takes a compressed sequence and expands it into the original sequence.

> **Hint:** You may want to use `my-append` and `replicate`.

`my-append` is implemented as follows, where `my-append` takes in two lists and concatenates them together.

```
(define (my-append a b)
    (if (null? a)
    b
    (cons (car a) (my-append (cdr a) b))))
```

```
scm> (my-append '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
```

```
 7        b
 8        (cons (car a) (my-append (cdr a) b))))
 9
10    (define (uncompress s)
11        (if (null? s)
12            nil
13            (my-append (replicate (car (car s)) (car (cdr (car s)))) (uncompress (cdr s))))
14    )
15
```

```
scm> (uncompress '((a 1) (b 2) (c 3)))
(a b b c c c)
```

Q8: (Tutorial) Map

Write a function that takes a procedure and applies it to every element in a given list.

```
(define (map fn lst)
    (cond ((null? lst) '())
          (else (cons (fn (car lst)) (map fn (cdr lst))))))
)

;;; Tests
(map (lambda (x) (* x x)) '(1 2 3))
; expect (1 4 9)

```

```
scm> (map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

Q9: (Tutorial) Make Tree

Fill in the following to complete an abstract tree data type:

```
(define (make-tree label branches) (cons label branches))

(define (label tree)
    (car tree)
)

(define (branches tree)
    (cdr tree)
)

```

Q10: (Tutorial) Tree Sum

Using the abstract data type above, write a function that sums up the entries of a tree, assuming that the entries are all numbers.

> **Hint:** you may want to use the `map` function you defined above, and also write a helper function for summing up the entries of a list.

```
(define (tree-sum tree)
  (cond
    ((null? tree)
     0)
    (else
     (+ (label tree) (tree-sum (branches tree))))))
```