CS 61A
Spring 2020

Interpreters

Discussion 10: April 8, 2020 Solutions

1 Calculator

An interpreter is a program that understands other programs. Today, we will explore how to interpret a simple language that uses Scheme syntax called *Calculator*.

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 given on the right. Recall that 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. `'+'`).

```
calc> (+ 2 2)
4

calc> (- 5)
-5

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

Call expressions are a bit more complicated. First, note that like Scheme call expressions, call expressions in Calculator 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
```

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(rest))
        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

1.1 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))
```

Box and pointers solutions
Video walkthrough

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

i. Write out the Python expression that returns a `Pair` representing the given expression, and draw a box and pointer diagram corresponding to it.

```
>>> Pair('+', Pair(Pair('-', Pair(2, Pair(4, nil))), Pair(6, Pair(8, nil))))
```

Box and pointer diagram

ii. 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?

`p.first`

iii. What are the operands of the call expression? If the `Pair` you constructed in Part (i) 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?

`p.rest` to get a list containing all the operands. `p.rest.first` to get the first operand by itself.

2 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. For example, the numbers 3.14 and 165 just evaluate to themselves.
2. **Names** are looked up in the `OPERATORS` 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
```

Note that `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 is more complex when it comes to applying Scheme procedures, which 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 evalutation rules take place.

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

Questions

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

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

6 calls to eval: 1 for the entire expression, and then 1 each for the operator and each operand.

1 call to apply the addition operator.

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

10 calls to eval: 1 for the whole expression, then 1 for each of the operators and operands. When we encounter another call expression, we have to evaluate the operators and operands inside as well.

3 calls to apply the function to the arguments for each call expression.
Video walkthrough

2.2 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?

Comparison expressions are regular call expressions, so we need to evaluate the operator and operands and then apply a function to the arguments. Therefore, we do not need to change `calc_eval`. We simply need to add new entries to the `OPERATORS` dictionary that map `'<'`, `'>'`, and `'='` to functions that perform the appropriate comparison operation.

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

Since `and` is a special form that short circuits on the first false-y operand, we cannot handle these expressions the same way we handle call expressions. We need to add special handling for combinations that don't evaluate all the operands.

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.

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        if _________________________: # and expressions
```

```
            return eval_and(exp.rest)
        else:                       # Call expressions
            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):


def calc_eval(exp):
    if isinstance(exp, Pair):
        if exp.first == 'and':  # and expressions
            return eval_and(exp.rest)
        else:                   # Call expressions
            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):
    curr, val = operands, True
    while curr is not nil:
        val = calc_eval(curr.first)
        if val is False:
            return False
        curr = curr.rest
    return val
```

3 List Questions

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

```
(define (replicate x n)


    (if (= n 0)
        nil
        (cons x (replicate x (- n 1)))))
```

Video walkthrough

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

3.2 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`.

Recall `my-append` is 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)
```

```
(define (uncompress s)


  (if (null? s)
    s
    (my-append (replicate (car (car s)) (car (cdr (car s))))
            (uncompress (cdr s)))))
```

Video walkthrough

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

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

```
(define (map fn lst)


    (if (null? lst)
        nil
        (cons (fn (car lst)) (map fn (cdr lst)))))

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

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

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

(define (label tree)

(define (branches tree)


(define (label tree) (car tree))
(define (branches tree) (cdr tree))
```

3.5 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)


    (+ (label tree) (sum (map tree-sum (branches tree)))))



(define (sum lst)
    (if (null? lst) 0 (+ (car lst) (sum (cdr lst)))))
```

1. **. . . That Factors Into Your Learning**

Implement the `factors` procedure in Scheme, which takes an integer `n` that is greater than `1` and returns a list of all of the factors of `n` from `1` to `n - 1` *in increasing order*. You may not need to use all the lines.

Hint: The built-in `modulo` procedure returns the remainder when dividing one number by the other.

```
scm> (modulo 5 3)
2
scm> (modulo 14 2)
0
```

```
(define (factors n)

  (define (factors-helper i n)

    (if (= i n)

        nil

        (if (= (modulo n i) 0)

            (cons i (factors-helper (+ i 1) n))

            (factors-helper (+ i 1) n)

        )))

  (factors-helper 1 n)

)
```

```
scm> (factors 6)
(1 2 3)
scm> (factors 7)
(1)
scm> (factors 28)
(1 2 4 7 14)
```

2. **A Deep Problem**

`deep-squares`, which takes a deep list of numbers and returns a list with each value squared, is given below.

```
1 (define (deep-squares lol)
2   (cond ((null? lol) '())
3         ((list? (car lol))
4          (cons (map square (car lol))
5                (deep-squares (cdr lol)) ))
6         (cons (square (car lol)) (deep-squares (cdr lol)) )))
```

For which of the following inputs will `deep-squares` not work as intended?

(a) `(deep-squares '())` ● Works ○ Broken

(b) `(deep-squares '(1 (2 3) 4))` ○ Works ● Broken

(c) `(deep-squares '(1 (2 3) ((4)) 5))` ○ Works ● Broken

Which line number(s) contain(s) the bug(s)? ◯ 1 ◯ 2 ◯ 3 ● 4 ◯ 5 ● 6

The final clause in the cond is missing a set of parentheses! (the word `else` is optional; we just need to enclose line 6 in one more pair of parentheses). Also, line 4 should be `(cons (deep-squares (car lol))`