CS 61A
Fall 2021

Interpreters

Discussion 11: November 10, 2021 Solutions

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. `'+'`).

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 the Python code, `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(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

Q1: 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:

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

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`

What are the operands of the call expression?

An expression (- 2 4), the number 6, the number 8.

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

`p.rest`

How would you retrieve only the first operand?

`p.rest.first`

Q2: New Procedure

Suppose we want to add the `//` operation to our Calculator interpreter. Recall from Python that `//` is the floor division operation, so we are looking to add a built-in procedure `//` in our interpreter such that `(// dividend divisor)` returns dividend // divisor. Similarly we handle multiple inputs as illustrated in the following example `(// dividend divisor1 divisor2 divisor3)` evaluates to (((dividend // divisor1) // divisor2) // divisor3). For this problem you can assume you are always given at least 1 divisor. Also for this question do you need to call `calc_eval` inside `floor_div`? Why or why not?

```
calc> (// 1 1)
1
calc> (// 5 2)
2
calc> (// 28 (+ 1 1) 1)
14
```

```
def calc_eval(exp):
    if isinstance(exp, Pair): # Call expressions
        return calc_apply(calc_eval(exp.first), exp.rest.map(
    calc_eval))
    elif exp in OPERATORS:      # Names
        return OPERATORS[exp]
    else:                       # Numbers
        return exp
def floor_div(expr):
    divident = expr.first
    expr = expr.rest
    while expr.rest != nil:
        divisor = expr.first
        dividend //= divisor
        expr = expr.rest
    return dividend

# Assume OPERATORS['//'] = floor_div is added for you in the code
```

Q3: New Form

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?

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

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 exp.first == 'and': # and expressions
            return eval_and(exp.rest)
        else:                  # Call expressions
            return calc_apply(calc_eval(exp.first), 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
```

Q4: Saving Values

In the last few questions we went through a lot of effort to add operations so we can do most arithmetic operations easily. However it's a real shame we can't store these values. So for this question let's implement a `define` special form that saves values to variable names. This should work like variable assignment in Scheme; this means that you should expect inputs of the form `(define <variable_name> <value>)` and these inputs should return the symbol corresponding to the variable name.

```
calc> (define a 1)
a
calc> a
1
```

This is a more involved change. Here are the 4 steps involved: 1. Add a `bindings` dictionary that will store the names and correspondings values of variables as key-value pairs of the dictionary. 2. Identify when the define form is given to

`calc_eval`. 3. Allow variables to be looked up in `calc_eval`. 4. Write the function `eval_define` which should actually handle adding names and values to the bindings dictionary.

We've done step 1 for you. Now you'll do the remaining steps in the code below.

```
bindings = {}
def calc_eval(exp):
    if isinstance(exp, Pair):
        if exp.first == 'and': # and expressions
            return eval_and(exp.rest)
        elif exp.first == 'define': # and expressions
            return eval_define(exp.rest)

        else:                    # Call expressions
            return calc_apply(calc_eval(exp.first), exp.rest.map(
    calc_eval))
    elif exp in bindings: # Looking up variables
        return bindings[exp]
    elif exp in OPERATORS:      # Looking up procedures
        return OPERATORS[exp]
    else:                       # Numbers
        return exp

def eval_define(expr):
    name, value = expr.first, calc_eval(expr.rest.first)
    bindings[name] = value
    return name
```

Q5: Counting Eval and Apply

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

```
scm> (+ 1 2)
```

For this particular prompt please list out the inputs to `calc_eval` and `calc_apply`.

4 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.

Explicity listing out the inputs we have the following for calc_eval: , '+', 1, 2. calc_apply is given '+' for fn and (1 2) for args.

A note is that (+ 1 2) corresponds to the following Pair, Pair('+', Pair(1, Pair(2, nil))) and (1 2) corresponds to the Pair, Pair(1, Pair(2, nil)).

```
scm> (+ 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.

```
scm> (+ 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.

```
scm> (and 1 (+ 1 0) 0)
```

7 calls to eval: 1 for the whole expression, 1 for the first argument, 1 for `(+ 1 0)`, 1 for the `+` operator, 2 for the operands to plus, and 1 for the final 0. Notice that and is a special form so we do not run `calc_eval` on the and.

1 calls to apply to evaluate the `+` expression.

Video Walkthrough

Q6: From Pair to Calculator

Write out the Calculator expression with proper syntax that corresponds to the following `Pair` constructor calls.

```
>>> 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