Lab 11: Interpreters **lab11.zip (lab11.zip)**

Due by 11:59pm on Wednesday, November 8.

Starter Files

Download lab11.zip (lab11.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Interpreters

Required Questions

Getting Started Videos

Calculator

An interpreter is a program that executes programs. Today, we will extend the interpreter for Calculator, a simple made-up language that is a subset of Scheme. This lab is like Project 4 in miniature.

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

```
calc> (+ 2 2 2)
6

calc> (- 5)
-5

calc> (* (+ 1 2) (+ 2 3 4))
27
```

Calculator expressions are represented as Python objects:

- Numbers are represented using Python numbers.
- The symbols for arithmetic operations are represented using Python strings (e.g. `'+'`).
- Call expressions are represented using the `Pair` class below.

Pair Class

To represent Scheme lists in Python, we will use the `Pair` class (in both this lab and the Scheme project). A `Pair` instance has two attributes: `first` and `rest`. `Pair` is always called on two arguments. To make a list, nest calls to `Pair` and pass in `nil` as the second argument of the last pair.

- Look familiar? `Pair` is very similar to `Link`, the class we used to represent linked lists. They differ in their `str` representation: printing a `Pair` instance displays the list using Scheme syntax.

> **Note** In the Python code, `nil` is bound to a user-defined object that represents an empty Scheme list. Similarly, `nil` in Scheme evaluates to an empty list.

For example, once our interpreter reads in the Scheme expression `(+ 2 3)`, it is represented as `Pair('+', Pair(2, Pair(3, nil)))`.

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

The `Pair` class has a `map` method that takes a one-argument python function `fn`. It returns the Scheme list that results from applying `fn` to each element of the Scheme list.

```
>>> p.rest.map(lambda x: 2 * x)
Pair(4, Pair(6, nil))
```

Pair Class

Q1: Using Pair

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

Use Ok to test your understanding:

```
python3 ok -q using_pair -u
```

Calculator Evaluation

For Question 2 (New Procedure) and Question 4 (Saving Values), you'll need to update the `calc_eval` function below, which evaluates a Calculator expression. For Question 2, you'll determine what are the `operator` and `operands` for a call expression in Scheme as well as how to apply a procedure to arguments the `calc_apply` line. For Question 4, you'll determine how to look up the value of symbols previously defined.

```
def calc_eval(exp):
    """
    >>> calc_eval(Pair("define", Pair("a", Pair(1, nil))))
    'a'
    >>> calc_eval("a")
    1
    >>> calc_eval(Pair("+", Pair(1, Pair(2, nil))))
    3
    """
    if isinstance(exp, Pair):
        operator = ____________ # UPDATE THIS FOR Q2
        operands = ____________ # UPDATE THIS FOR Q2
        if operator == 'and': # and expressions
            return eval_and(operands)
        elif operator == 'define': # define expressions
            return eval_define(operands)
        else: # Call expressions
            return calc_apply(___________, ___________) # UPDATE THIS FOR Q2
    elif exp in OPERATORS:   # Looking up procedures
        return OPERATORS[exp]
    elif isinstance(exp, int) or isinstance(exp, bool):   # Numbers and booleans
        return exp
    elif _________________: # CHANGE THIS CONDITION FOR Q4
        return _________________ # UPDATE THIS FOR Q4
```

Q2: New Procedure

Add the `//` operation to Calculator, a floor-division procedure such that `(// dividend divisor)` returns the result of dividing `dividend` by `divisor`, ignoring the remainder (`dividend // divisor` in Python). Handle multiple inputs as illustrated in the following example: `(// dividend divisor1 divisor2 divisor3)` evaluates to `(((dividend // divisor1) // divisor2) // divisor3)` in Python. Assume every call to `//` has at least two arguments.

> *Hint:* You will need to modify both the `calc_eval` and `floor_div` methods for this question!

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

> *Hint:* Make sure that every element in a `Pair` (the operator and all operands) will be `calc_eval` -uated once, so that we can correctly apply the relevant Python operator to operands! You may find the `map` method of the `Pair` class useful for this.

```
def floor_div(args):
    """
    >>> floor_div(Pair(100, Pair(10, nil)))
    10
    >>> floor_div(Pair(5, Pair(3, nil)))
    1
    >>> floor_div(Pair(1, Pair(1, nil)))
    1
    >>> floor_div(Pair(5, Pair(2, nil)))
    2
    >>> floor_div(Pair(23, Pair(2, Pair(5, nil))))
    2
    >>> calc_eval(Pair("//", Pair(4, Pair(2, nil))))
    2
    >>> calc_eval(Pair("//", Pair(100, Pair(2, Pair(2, Pair(2, Pair(2, Pair(2, nil)))))))
    3
    >>> calc_eval(Pair("//", Pair(100, Pair(Pair("+", Pair(2, Pair(3, nil))), nil))))
    20
    """
    # BEGIN SOLUTION Q2
```

Use Ok to test your code:

```
python3 ok -q floor_div
```

Q3: New Form

Add `and` expressions to our Calculator interpreter as well as introduce the Scheme boolean values `#t` and `#f`, represented as Python `True` and `False`. (The examples below assumes conditional operators (e.g. `<`, `>`, `=`, etc) have already been implemented, but you do not have to worry about them for this question.)

```
calc> (and (= 1 1) 3)
3
calc> (and (+ 1 0) (< 1 0) (/ 1 0))
#f
calc> (and #f (+ 1 0))
#f
calc> (and 0 1 (+ 5 1)) ; 0 is a true value in Scheme!
6
```

In a call expression, we first evaluate the operator, then evaluate the operands, and finally apply the procedure to its arguments (just like you did for `floor_div` in the previous question). However, since `and` is a special form that short circuits on the first false argument, we cannot evaluate `and` expressions the same way we evaluate call expressions. We need to add special logic for forms that don't always evaluate all the sub-expressions.

```
scheme_t = True   # Scheme's #t
scheme_f = False  # Scheme's #f

def eval_and(expressions):
    """
    >>> calc_eval(Pair("and", Pair(1, nil)))
    1
    >>> calc_eval(Pair("and", Pair(False, Pair("1", nil))))
    False
    >>> calc_eval(Pair("and", Pair(1, Pair(Pair("//", Pair(5, Pair(2, nil))), nil))))
    2
    >>> calc_eval(Pair("and", Pair(Pair('+', Pair(1, Pair(1, nil))), Pair(3, nil))))
    3
    >>> calc_eval(Pair("and", Pair(Pair('-', Pair(1, Pair(0, nil))), Pair(Pair('/', Pair(5
    2.5
    >>> calc_eval(Pair("and", Pair(0, Pair(1, nil))))
    1
    >>> calc_eval(Pair("and", nil))
    True
    """
    # BEGIN SOLUTION Q3
```

Use Ok to test your code:

```
python3 ok -q eval_and
```

Q4: Saving Values

Implement a `define` special form that binds values to symbols. This should work like `define` in Scheme: `(define <symbol> <expression>)` first evaluates the `expression`, then binds the `symbol` to its value. The whole `define` expression evaluates to the `symbol`.

```
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 symbols and correspondings values (done for you).
2. Identify when the define form is given to `calc_eval` (done for you).
3. Allow symbols bound to values to be looked up in `calc_eval`.
4. Write the function `eval_define` which should add symbols and values to the bindings dictionary.

```
bindings = {}

def eval_define(expressions):
    """
    >>> eval_define(Pair("a", Pair(1, nil)))
    'a'
    >>> eval_define(Pair("b", Pair(3, nil)))
    'b'
    >>> eval_define(Pair("c", Pair("a", nil)))
    'c'
    >>> calc_eval("c")
    1
    >>> calc_eval(Pair("define", Pair("d", Pair("//", nil))))
    'd'
    >>> calc_eval(Pair("d", Pair(4, Pair(2, nil))))
    2
    """
    # BEGIN SOLUTION Q4
```

Use Ok to test your code:

```
python3 ok -q eval_define
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Make sure to submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** For a refresher on how to do this, refer to Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope).