PyCombinator Interpreter

Download lab package, source of PyCombinator Interpreter。Follow the steps below to warm up before lab exercise.

0: read-eval-print

Read repl.py, try to remain yourself of how interpreter works. You can start the interpreter by running following cmd:

```
python repl.py
```

```
H:\lab10>python repl.py
>
```

Since our code is incomplete, the interpreter is still unusable. But once the exercises are done, your interpreter should be able to read cmd and print answer as follow:

```
> add(3, 4)
7
> mul(4, 5)
20
> sub(2, 3)
-1
> (lambda: 4)()
4
> (lambda x, y: add(y, x))(3, 5)
8
> (lambda x: lambda y: mul(x, y))(3)(4)
12
> (lambda f: f(0))(lambda x: pow(2, x))
1
```

1: read

Read reader.py, especially function **read**(s). Try to solve " ___ your answer ____" before running in IDE or cmd shell.

```
>>> read('sdhkahd')
    ___ your answer ____
>>> read('3')
    ___ your answer ____
>>> read('lambda f: f(0)')
    ___ your answer ____
```

```
>>> read('(lambda x: x)(5)')
    ___ your answer ____
>>> read('(lambda: 5)()')
    ___ your answer ____
>>> read('lambda x y: 10')
    ___ your answer ____
>>> read('lambda x : lambda y : 10')
    ___ your answer ____
>>> read('  ')
    ___ your answer ____
```

Then Use Ok to test your understanding of the reader.

```
python ok -q prologue_reader -u
```

```
C:\Users\[redacted]>cd H:\lab10

C:\Users\[redacted]>H:

H:\lab10>python ok -q prologue_reader -u
=====================================================================
Assignment: Lab 10
OK, version v1.14.19
=====================================================================
```

2: eval & print

Read expr.py, then Use Ok to test your understanding of the Expr and Value objects.

```
python3 ok -q prologue_expr -u
```

```
H:\lab10>python3 ok -q prologue_expr -u
=====================================================================
Assignment: Lab 10
OK, version v1.14.19
=====================================================================
```

Now the warm up is over! It is required that you should complete following exercises by modifying only expr.py. Submit to **sicp@foxmail.com** if accomplish. Due by **12/10.**

Exercise 1: Evaluating Names

The first type of PyCombinator expression that we want to evaluate are names. In our program, a name is an instance of the `Name` class. Each instance has a `string` attribute which is the name of the variable -- e.g. `"x"`.

Recall that the value of a name depends on the current environment. In our implementation, an environment is represented by a dictionary that maps variable names (strings) to their values (instances of the `Value` class).

The method `Name.eval` takes in the current environment as the parameter `env` and returns the value bound to the `Name`'s `string` in this environment. Implement it as follows:

- If the name exists in the current environment, look it up and return the value it is bound to.
- If the name does not exist in the current environment, raise a `NameError` with an appropriate error message:

```
raise NameError('your error message here (a string)')
```

`You are required to fill in the code of expr.py – class Name –` **`def eval(self, env).`**

Exercise 2: Evaluating Call Expressions

Now, let's add logic for evaluating call expressions, such as `add(2, 3)`. Remember that a call expression consists of an operator and 0 or more operands.

In our implementation, a call expression is represented as a `CallExpr` instance. Each instance of the `CallExpr` class has the attributes `operator` and `operands`. `operator` is an instance of `Expr`, and, since a call expression can have multiple operands, `operands` is a *list* of `Expr` instances.

For example, in the `CallExpr` instance representing `add(3, 4)`:

- `self.operator` would be `Name('add')`
- `self.operands` would be the list `[Literal(3), Literal(4)]`

In `CallExpr.eval`, implement the three steps to evaluate a call expression:

1. Evaluate the *operator* in the current environment.
2. Evaluate the *operand(s)* in the current environment.
3. Apply the value of the operator, a function, to the value(s) of the operand(s).

Hint: Since the operator and operands are all instances of `Expr`, you can evaluate them by calling their `eval` methods. Also, you can apply a function (an instance of `PrimitiveFunction` or `LambdaFunction`) by calling its `apply` method, which takes in a list of arguments (`Value` instances).

`You are required to fill in the code of expr.py – class CallExpr –` **`def eval(self, env).`**

Exercise 3: Applying Lambda Functions

We can do some basic math now, but it would be a bit more fun if we could also call our own user-defined functions. So let's make sure that we can do that!

A lambda function is represented as an instance of the `LambdaFunction` class. If you look in `LambdaFunction.__init__`, you will see that each lambda function has three instance attributes: `parameters`, `body` and `parent`. As an example, consider the lambda function `lambda f, x: f(x)`. For the corresponding `LambdaFunction` instance, we would have the following attributes:

- `parameters` -- a list of strings, e.g. `['f', 'x']`
- `body` -- an `Expr`, e.g. `CallExpr(Name('f'), [Name('x')])`
- `parent` -- the parent environment in which we want to look up our variables. Notice that this is the environment the lambda function was defined in. `LambdaFunction`s are created in the `LambdaExpr.eval` method, and the current environment then becomes this `LambdaFunction`'s parent environment.

If you try entering a lambda expression into your interpreter now, you should see that it outputs a lambda function. However, if you try to call a lambda function, e.g. `(lambda x: x)(3)` it will output `None`.

You are now going to implement the `LambdaFunction.apply` method so that we can call our lambda functions! This function takes a list `arguments` which contains the argument `Value`s that are passed to the function. When evaluating the lambda function, you will want to make sure that the lambda function's formal parameters are correctly bound to the arguments it is passed. To do this, you will have to modify the environment you evaluate the function body in.

There are three steps to applying a `LambdaFunction`:

1. Make a copy of the parent environment. You can make a copy of a dictionary `d` with `d.copy()`.
2. Update the copy with the `parameters` of the `LambdaFunction` and the `arguments` passed into the method.
3. Evaluate the `body` using the newly created environment.

`You are required to fill in the code of expr.py – class CallExpr –` **`def LambdaExpr(self, env).`**

Check your answer

`Check if your interpreter can run as shown in` **`0:read-eval-print`**

```
> add(3, 4)
7
> mul(4, 5)
20
> sub(2, 3)
-1
> (lambda: 4)()
4
> (lambda x, y: add(y, x))(3, 5)
8
> (lambda x: lambda y: mul(x, y))(3)(4)
12
> (lambda f: f(0))(lambda x: pow(2, x))
1
```

If so, submit your code(only expr.py) to sicp@foxmail.com with title: **lab08_学号_姓名**. Due by 12/10.