Lab 11: Interpreters

> Adapted from cs61a of UC Berkeley.

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-lab11.git

```
git clone https://github.com/JacyCui/sicp-lab11.git
```

`lab11.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip lab11.zip
```

`README.md` is the handout for this homework. `solution` is a probrab solution of the lab. However, I might not give my solution exactly when the lab is posted. You need to finish the task on your own first. If any problem occurs, please make use of the comment section.

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

An interpreter is a program that allows you to interact with the computer in a certain language. It understands the expressions that you type in through that language, and performs the corresponding actions in some way, usually using an underlying language.

In Project 4, you will use Python to implement an interpreter for Scheme. The Python interpreter that you've been using all semester is written (mostly) in the C programming language. The computer itself uses hardware to interpret machine code (a series of ones and zeros that represent basic operations like adding numbers, loading information from memory, etc).

When we talk about an interpreter, there are two languages at work:

1. **The language being interpreted/implemented.** In this lab, you will implement the PyCombinator language.
2. **The underlying implementation language.** In this lab, you will use Python to implement the PyCombinator language.

Note that the underlying language need not be different from the implemented language. In fact, in this lab we are going to implement a smaller version of Python (PyCombinator) using Python! This idea is called Metacircular Evaluation.

Many interpreters use a Read-Eval-Print Loop (REPL). This loop waits for user input, and then processes it in three steps:

- **Read:** The interpreter takes the user input (a string) and passes it through a lexer and parser.
  - The *lexer* turns the user input string into atomic pieces (tokens) that are like "words" of the implemented language.
  - The *parser* takes the tokens and organizes them into data structures that the underlying language can understand.
- **Eval:** Mutual recursion between eval and apply evaluate the expression to obtain a value.
  - *Eval* takes an expression and evaluates it according to the rules of the language. Evaluating a call expression involves calling `apply` to apply an evaluated operator to its evaluated operands.
  - *Apply* takes an evaluated operator, i.e., a function, and applies it to the call expression's arguments. Apply may call `eval` to do more work in the body of the function, so `eval` and `apply` are *mutually recursive*.
- **Print:** Display the result of evaluating the user input.

Here's how all the pieces fit together:

```
         +-------------------------------- Loop -----------+
         |                                                 |
         |  +-------+   +--------+   +-------+   +-------+ |
Input ---+->| Lexer |-->| Parser |-->| Eval  |-->| Print |-+--> Output
         |  +-------+   +--------+   +-------+   +-------+ |
         |                              ^  |               |
         |                              |  v               |
         ^                           +-------+             v
         |                           | Apply |             |
         |    REPL                   +-------+             |
         +-------------------------------------------------+
```

Required Questions

PyCombinator Interpreter

Today we will build **PyCombinator**, our own basic Python interpreter. By the end of this lab, you will be able to use a bunch of primitives such as `add`, `mul`, and `sub`, and even more excitingly, we will be able to create and call lambda functions -- all through your own homemade interpreter!

You will implement some of the key parts that will allow us to evaluate the following commands and more:

```
> 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
```

You can find the Read-Eval-Print Loop code for our interpreter in `repl.py`. Here is an overview of each of the REPL components:

- **Read:** The function `read` in `reader.py` calls the following two functions to parse user input.
  - The *lexer* is the function `tokenize` in `reader.py` which splits the user input string into tokens.
  - The *parser* is the function `read_expr` in `reader.py` which parses the tokens and turns expressions into instances of subclasses of the class `Expr` in `expr.py`, e.g. `CallExpr`.
- **Eval:** Expressions (represented as `Expr` objects) are evaluated to obtain values (represented as `Value` objects, also in `expr.py`).
  - *Eval*: Each type of expression has its own `eval` method which is called to evaluate it.
  - *Apply*: Call expressions are evaluated by calling the operator's `apply` method on the arguments. For lambda procedures, `apply` calls `eval` to evaluate the body of the function.
- **Print:** The `__str__` representation of the obtained value is printed.

In this lab, you will only be implementing the *Eval* and *Apply* steps in `expr.py`.

You can start the PyCombinator interpreter by running the following command:

```
python3 repl.py
```

Try entering a literal (e.g. `4`) or a lambda expression, (e.g. `lambda x, y: add(x, y)`) to see what they evaluate to.

You can also try entering some names. You can see the entire list of names that we can use in PyCombinator at the bottom of `expr.py`. Note that our set of primitives doesn't include the operators `+`, `-`, `*`, `/` -- these are replaced by `add`, `sub`, etc.

Right now, any names (e.g. `add`) and call expressions (e.g. `add(2, 3)`) will output `None`. It's your job to implement `Name.eval` and `CallExpr.eval` so that we can look up names and call functions in our interpreter!

You don't have to understand how the read component of our interpreter is implemented, but if you want a better idea of how user input is read and transformed into Python code, you can use the `--read` flag when running the interpreter:

```
$ python3 repl.py --read
> add
Name('add')
> 3
Literal(3)
> lambda x: mul(x, x)
LambdaExpr(['x'], CallExpr(Name('mul'), [Name('x'), Name('x')]))
> add(2, 3)
CallExpr(Name('add'), [Literal(2), Literal(3)])
```

To exit the interpreter, type Ctrl-C or Ctrl-D.

For this lab, you will be writing code in `expr.py`.

Q1: Prologue

Before we write any code, let's try to understand the parts of the interpreter that are already written.

Here is the breakdown of our implementation:

- `repl.py` contains the logic for the REPL loop, which repeatedly reads expressions as user input, evaluates them, and prints out their values (you don't have to completely understand all the code in this file).
- `reader.py` contains our interpreter's reader. The function `read` calls the functions `tokenize` and `read_expr` to turn an expression string into an `Expr` object (you don't have to completely understand all the code in this file).
- `expr.py` contains our interpreter's representation of expressions and values. The subclasses of `Expr` and `Value` encapsulate all the types of expressions and values in the PyCombinator language. The global environment, a dictionary containing the bindings for primitive functions, is also defined at the bottom of this file.

> Use Ok to test your understanding of the reader. It will be helpful to refer to `reader.py` to answer these questions.
>
> ```
> python3 ok -q prologue_reader -u --local
> ```
>
> Use Ok to test your understanding of the `Expr` and `Value` objects. It will be helpful to refer to `expr.py` to answer these questions.
>
> ```
> python3 ok -q prologue_expr -u --local
> ```

Q2: 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 `var_name` 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 `var_name` 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, return `None`

You will add code to the file `expr.py`.

```
def eval(self, env):
    """
    >>> env = {
    ...     'a': Number(1),
    ...     'b': LambdaFunction([], Literal(0), {})
    ... }
    >>> Name('a').eval(env)
    Number(1)
    >>> Name('b').eval(env)
    LambdaFunction([], Literal(0), {})
    >>> print(Name('c').eval(env))
    None
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q Name.eval --local
```

Now that you have implemented the evaluation of names, you can look up names in the global environment like `add` and `sub` (see the full list of primitive math operators in `global_env` at the bottom of `expr.py`). You can also try looking up undefined names to see how the `NameError` is displayed!

```
$ python3 repl.py
> add
<primitive function add>
```

Unfortunately, you still cannot call these functions. We'll fix that next!

Q3: 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 will add code to the file `expr.py`.

```python
def eval(self, env):
    """
    >>> from reader import read
    >>> new_env = global_env.copy()
    >>> new_env.update({'a': Number(1), 'b': Number(2)})
    >>> add = CallExpr(Name('add'), [Literal(3), Name('a')])
    >>> add.eval(new_env)
    Number(4)
    >>> new_env['a'] = Number(5)
    >>> add.eval(new_env)
    Number(8)
    >>> read('max(b, a, 4, -1)').eval(new_env)
    Number(5)
    >>> read('add(mul(3, 4), b)').eval(new_env)
    Number(14)
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q CallExpr.eval --local
```

Now that you have implemented the evaluation of call expressions, we can use our interpreter for simple expressions like `sub(3, 4)` and `add(mul(4, 5), 4)`. Open your interpreter to do some cool math:

```
$ python3 repl.py
```

Optional Questions

Q4: 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.

> *Hint:* You may find the built-in `zip` function useful to pair up the parameter names with the argument values.

```
def apply(self, arguments):
    """
    >>> from reader import read
    >>> add_lambda = read('lambda x, y: add(x, y)').eval(global_env)
    >>> add_lambda.apply([Number(1), Number(2)])
    Number(3)
    >>> add_lambda.apply([Number(3), Number(4)])
    Number(7)
    >>> sub_lambda = read('lambda add: sub(10, add)').eval(global_env)
    >>> sub_lambda.apply([Number(8)])
    Number(2)
    >>> add_lambda.apply([Number(8), Number(10)]) # Make sure you made a copy of
env
    Number(18)
    >>> read('(lambda x: lambda y: add(x, y))(3)(4)').eval(global_env)
    Number(7)
    >>> read('(lambda x: x(x))(lambda y: 4)').eval(global_env)
    Number(4)
    """
    if len(self.parameters) != len(arguments):
        raise TypeError("Oof! Cannot apply number {} to arguments {}".format(
            comma_separated(self.parameters), comma_separated(arguments)))
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q LambdaFunction.apply --local
```

After you finish, you should try out your new feature! Open your interpreter and try creating and calling your own lambda functions. Since functions are values in our interpreter, you can have some fun with higher order functions, too!

```
$ python3 repl.py
> (lambda x: add(x, 3))(1)
4
> (lambda f, x: f(f(x)))(lambda y: mul(y, 2), 3)
12
```

Q5: Handling Exceptions

The interpreter we have so far is pretty cool. It seems to be working, right? Actually, there is one case we haven't covered. Can you think of a very simple calculation that is undefined (maybe involving division)? Try to see what happens if you try to compute it using your interpreter (using `floordiv` or `truediv` since we don't have a standard `div` operator in PyCombinator). It's pretty ugly, right? We get a long error message and exit our interpreter -- but really, we want to handle this elegantly.

Try opening up the interpreter again and see what happens if you do something ill defined like `add(3, x)`. We just get a nice error message saying that `x` is not defined, and we can then continue using our interpreter. This is because our code handles the `NameError` exception, preventing it from crashing our program. Let's talk about how to handle exceptions:

In lecture, you learned how to raise exceptions. But it's also important to catch exceptions when necessary. Instead of letting the exception propagate back to the user and crash the program, we can catch it using a `try/except` block and allow the program to continue.

```
try:
    <try suite>
except <ExceptionType 0> as e:
    <except suite 0>
except <ExceptionType 1> as e:
    <except suite 1>
...
```

We put the code that might raise an exception in the `<try suite>`. If an exception is raised, then the program will look at what type of exception was raised and look for a corresponding `<except suite>`. You can have as many except suites as you want.

```
try:
    1 + 'hello'
except NameError as e:
    print('hi')  # NameError except suite
except TypeError as e:
    print('bye') # TypeError except suite
```

In the example above, adding `1` and `'hello'` will raise a `TypeError`. Python will look for an except suite that handles `TypeError`s -- the second except suite. Generally, we want to specify exactly which exceptions we want to handle, such as `OverflowError` or `ZeroDivisionError` (or both!), rather than handling all exceptions.

Notice that we can define the exception `as e`. This assigns the exception object to the variable `e`. This can be helpful when we want to use information about the exception that was raised.

```
>>> try:
...     x = int("cs61a rocks!")
... except ValueError as e:
...     print('Oops! That was no valid number.')
...     print('Error message:', e)
```

You can see how we handle exceptions in your interpreter in `repl.py`. Modify this code to handle ill-defined arithmetic errors, as well as type errors. Go ahead and try it out!