Prof. Jingke Li (lij@pdx.edu), Classes: TR 1400-1550 @OND 218, Office Hours: TR 1300-1400 (in-person @FAB 120-06). TAs: Nicholas Coltharp (`coltharp`), M 2-4pm @Fishbowl; Laura Israel (`laisrael`), W 10-11 @Fishbowl, F 1-2pm (Zoom: 761-457-3865)

Assignment 4 Handling Functions

(Due Thursday 12/5/24)

This last assignment adds one more feature, functions, to our simple imperative language, ImpLang. The new language is called ToyLang. You'll implement interpreters for two versions of this new language. (A third, optional version is also available for extra points.) The assignment carries a total of 20 points.

1. [*8 points*] An Interpreter for ToyLang

The first version of ToyLang has the following grammar:

```
prog -> stmt

stmt -> "var" ID "=" expr
      | "print" "(" expr ")"
      | "{" stmt (";" stmt)* "}"

expr -> "lambda" ID ":" expr     # function definition
      | expr "(" expr ")"        # function application
      | aexpr
aexpr -> aexpr "+" term
       | aexpr "-" term
       | term
term -> term "*" atom
      | term "/" atom
      | atom
atom: "(" expr ")"
      | ID
      | NUM
```

This new language requires a subset of ImpLang's features: variable declaration, print statement, statement block, and the expression section. (*Note:* You may augment this language by adding more ImpLang features (such as other statements) if you feel confident about their implementation in your previous interpreters.)

The two new features are function definition and function application. Both are defined as expressions.

A function definition takes the form of a lambda expression, with a formal parameter and a body expression:

```
"lambda" ID ":" expr
```

Evaluating a lambda-expression `"lambda x: body"` creates a **closure** `(x, body, env)`, where `x` and `body` are verbatim copies of the lambda expression's components (i.e. without evaluation), and `env` is a copy of the current environment, within which the function is defined.

Closures represent function values. In ToyLang, they are first-class values: they can be assigned to variables, passed as parameters to functions, and returned as function body's values.

When a lambda expression is assigned to a variable, the corresponding `var-closure` binding can be stored into the environment, just like a `var-integer` binding. A lambda expression can also be used directly in a function call. (See below.)

In the interpreter, you should define a `Closure` class to represent closures:

```
class Closure():
    def __init__(self, id, body, env):
        self.id = id
        self.body = body
        self.env = env
```

A function application takes the form

```
expr "(" expr ")"
```

The first `expr` represents a function, and should evaluate to a closure. With ToyLang, this `expr` can be either a variable that binds to a closure or a lambda expression. The second `expr` represents an argument.

Evaluating a function call `"f(arg)"` is a little complicated; the interpreter needs to follow these steps:

- evaluate `f` to a closure `(x, body, env)`,
- evaluate `arg` to a value `argv`, and bind it to the formal parameter `x` of the function,
- use the closure's environment `env` as base, open a new scope, and add the binding `x->argv` to the new environment,
- evaluate the function `body` in the new environment to a value `val`,
- close the new scope, and revert the current environment to the caller's env,
- yield the value `val` as result.

There are two key issues in the evaluation of a function call: (1) The function body needs to be evaluated in the function definition's environment (which can be found in the closure), and (2) For this evaluation, a new scope needs to be opened to allow the formal parameter's binding be effective in it (and not to conflict with outer-scopes' bindings with the same name).

A set of ToyLang programs are provided for you to test your interpreter:

inc1.toy
```
{ var inc = lambda x: x+1;
  print(inc(1)) }
```

add1.toy
```
{ var add = lambda x: lambda y: x+y;
  print(add(1)(2)) }
```

inc2.toy
```
{ var x = 1;
  var inc = lambda x: x+1;
  print(inc(x)) }
```

add2.toy
```
{ var x = 1;
  var y = 2;
  var add = lambda x: lambda y: x+y;
  print(add(x)(y)) }
```

inc3.toy
```
print((lambda x: x+1)(1))
```

add3.toy
```
print((lambda x: lambda y: x+y)(1)(2))
```

twice.toy
```
{ var twice = lambda f: (lambda z: f(f(z)));
  var inc = lambda y: y+1;
  print(twice(inc)(1))
}
```

These examples show that ToyLang directly supports only single-argument functions, but through currying, it can also support multi-argument functions.

2. [*8 points*] An Interpreter for ToyLang2

The lambda function definition has a limitation — the function body can only be an expression. With this form, we can't define functions that needs multiple statements, such as the factorial function.

In the second version of our new language, ToyLang2, we add a function declaration to the statement section:

```
stmt -> "def" ID "(" ID ")" "=" body
body -> "{" (stmt ";")* "return" expr "}"
```

As shown, this new form of function definition allows statements in the body, yet it requires a value to be returned at the end. So a call to this new function form will always return a value, just like calling a lambda function. This means that the new feature has no impact on the function call part of the language.

The second addition to this new version is to restore three additional statements from ImpLang, assignment, if statement, and while loop. The full statement section of the new grammar is shown below:

```
stmt -> "var" ID "=" expr
      | "def" ID "(" ID ")" "=" body
      | ID "=" expr
      | "if" "(" expr ")" stmt ["else" stmt]
      | "while" "(" expr ")" stmt
      | "print" "(" expr ")"
      | "{" stmt (";" stmt)* "}"
```

Evaluating a function declaration `"def f(x) = body"` is generally equivalent to evaluating a variable declaration of the form `"var f = lambda x: body"`, except that `body` is not a single expression:

- create a closure `(x, body, env)` and binds it to `f`,
- add the binding `f->closure` to the current environment.

Call your ToyLang2 interpreter `toylang2.py`. All test programs for the first version should also work for this version. In addition, several `.toy2` programs are available for testing the new function declaration feature.

3. [*2 points*] Programming in ToyLang2

Write two ToyLang2 programs, and validate them with your interpreter:

- **sum.toy2** — Define a function `sum(n)` to sum up values from 1 to `n`, and return the sum. The program then prints out two calls to this function: `sum(10)` and `sum(100)`.
- **fac.toy2** — Define a factorial function `fac(n)`, and print out two calls to this function: `fac(5)` and `fac(10)`.

4. [*Optional, 4 points*] An Interpreter for ToyLang3

The third version of ToyLang extends function declarations to allow multiple arguments:

```
stmt -> "def" ID "(" idlist ")" "=" body
idlist ->: ID ("," ID)*
```

Correspondingly, the function call syntax also need to change:

```
expr -> expr "(" explist ")"
explist -> expr ("," expr)*
```

So does the `Closure` class definition in the interpreter:

```
class Closure():
    def __init__(self, ids, body, env):
        self.ids = ids
        self.body = body
        self.env = env
```

The main change needed in the interpreter is for the function call. Instead of a single `param-argv` binding, multiple such bindings need to added to the new environment for evaluating a function's body.

If you choose to work on this part, include a simple test program, e.g. rewrite `add.toy` to use a true two-argument function.

5. [*2 points*] Summary Report

Write a short summary covering your experience with this assignment, including the following:

- Status of your programs. Do they successfully run on all tests you conducted? If not, describe the remaining issues as clearly as possible.
- Experience and lessons. What issues did you encounter? How did you resolve them?

Save your summary in a text or pdf file; call it `report.[txt|pdf]`.

Submission

Zip the interpreter programs, `toylang*.py`, the two Toylang2 programs, and the summary report, into a single zip file `assign4sol.zip`, and upload it through the upload it through the "File Upload" tab in the "Assignment 4" folder. (You need to press the "Start Assignment" button to see the submission options.) *Important:* Keep a copy of your submission file in your folder, and do not touch it. In case there is a glitch in Canvas submission system, the time-stamp on this file will serve as a proof of your original submission time.