Lab 5: Functions

Learning objectives:

- Understand how higher-order functions are used
- Understand different function-calling conventions
- Understand shallow versus deep binding
- Gain experience writing-higher order functions

You will make use of higher-order functions in both homework 2 and project 3, and as such the aim of this lab is to give you practice using them.

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab05/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Higher-order functions in Python.* Starter code for this problem can be found in `hof.py`. To run the doctests on your implementation:

   ```
   $ python3 -m doctest hof.py
   ```

   a) Implement the `make_stringbuilder()` higher-order function, which returns a "string-builder" function. A string-builder function maintains a current string, and calling the string-builder function should return the current string with the argument stringified and appended to it.

   ```
   >>> sb = make_stringbuilder()
   >>> sb('string2')
   'string2'
   >>> sb(100)
   'string2100'
   ```

   `make_stringbuilder()` also takes an optional parameter, and if present the current string should be initialized to the string form of the argument.

   ```
   >>> sb2 = make_stringbuilder('hello')
   >>> sb2(True)
   'helloTrue'
   ```

   Calling the string builder function with no parameter should return the current string.

   ```
   >>> sb2()
   'helloTrue'
   ```

   Use the built-in `str()` constructor to stringify an object.

   b) Implement the `trace()` decorator in Python. The decorator takes in a function and returns a version of that function that prints information about the function call every time it is called.

   ```
   >>> @trace
   ... def func(a, b):
   ... return a + b
   ...
   >>> func(1, 2)
   func(1, 2)
   3
   >>> @trace
   ... def bar(*args, **kwargs):
   ... print(args[0])
   ```

```
...
>>> bar(3, 4, foo='hello', baz='world')
bar(3, 4, foo='hello', baz='world')
3
```

Use the built-in `repr()` function to stringify an argument. (See this page and this post for the differences between `str()` and `repr()`.)

Hint: for any function `fn`, `fn.__name__` is the name of the function.

2. *Higher-order functions in Scheme.* Implement the `trace` higher-order procedure in Scheme. Much like the `trace()` Python decorator, this procedure should take in a function and return a version of that function that prints information about the function call every time it is called.

```
> (define (factorial n)
    (if (<= n 1)
        1
        (* n (factorial (- n 1)))
    )
  )
> (define factorial (trace factorial))
> (factorial 5)
(#<procedure:factorial> 5)
(#<procedure:factorial> 4)
(#<procedure:factorial> 3)
(#<procedure:factorial> 2)
(#<procedure:factorial> 1)
120
> (define add (trace +))
> (add 2 3 5 7)
(#<procedure:+> 2 3 5 7)
17
```

When printing out function arguments, use the `write` procedure instead of `display`. This is so that strings are printed out with quotation marks and escape sequences. For the purpose of this question, you are allowed to use `define` at a non-global level (since we haven't covered anonymous functions yet).

You can use the built-in `apply` procedure to apply a given procedure to a list of arguments:

```
> (apply + '(3 4 5))
12
```

Starter code for this problem can be found in `trace.scm`. To test your implementation:

```
$ plt-r5rs trace.scm > trace.out
$ diff trace.out trace.correct
```

3. *Calling conventions.* Consider the following code in a language with C-like syntax. Assume the `print()` function prints a value to standard output, that `main()` is the entry point of the program, and that + on strings denotes concatenation.

```
void func(string arg1, string arg2) {
  arg1 = arg1 + arg2;
  arg2 = arg2 + arg1;
}

int main() {
  string spam = "spam";
  string egg = "egg";
  func(spam, egg);
  print(spam);
  print(egg);
}
```

a) Determine the output of the code if the language were to use **call by value**.

b) Determine the output of the code if the language were to use **call by reference**.

c) Determine the output of the code if the language were to use **call by value-result**.

d) How many copies are done when the program uses call by value? Call by reference? What about call by value-result? (Assume the string concatenation operation doesn't do any copies)

4. *Binding policy.* Consider the following code in a language with C-like syntax. Assume the `print()` function prints a value to standard output and that `main()` is the entry point of the program.

```
void foo(void (*fun)()) {
  int x = 7;
  print(y);
  fun();
}

void bar() {
  print(x + y);
}

// fun is a function pointer to a function that
// takes in another function pointer as a parameter
void func(void (*fun)(void (*)())) {
  int y = 8;
  fun(bar);
}

int main() {
  int y = 3, x = 5;
  func(foo);
}
```

a) Determine the output of the code if the language were to use dynamic scope with **shallow binding**.

b) Determine the output of the code if the language were to use dynamic scope with **deep binding**.