Lab 2: Python and Value/Reference Semantics

Learning objectives:

- Understand the difference between reference semantics and value semantics
- Review writing recursive code
- Gain experience reading a language specification

Most of the code you write in this course will be in Python or Scheme, both of which have reference semantics. This lab will help you adjust to that. This lab also gives you practice in writing recursive code as well as reading a language specification, both of which will be necessary for projects 2 through 5.

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

```
$ wget https://eecs390.github.io/lab/lab02/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Value and reference semantics.* Consider two fragments of code in C++ and Python:

C++	**Python**
`class Foo { ... };`	`class Foo:`
	`...`
`int main() {`	
`Foo x = Foo();`	`x = Foo()`
`Foo y = x;`	`y = x`
`Foo &z = y;`	`z = y`
`x = Foo();`	`x = Foo()`
`}`	

For each program, draw a memory diagram that represents the state of the program just before it terminates.

2. *Reference semantics and swap.* Alyssa P. Hacker is trying to write a Python function to swap the contents of two lists. Here is what she has tried so far:

```
def swap_contents(list1, list2):
    """Modify list1 to contain the contents of list2 and vice versa.

    >>> list1 = [1, 2, 3]
    >>> list2 = [’hello’, ’world’]
    >>> swap_contents(list1, list2)
    >>> list1
    [’hello’, ’world’]
    >>> list2
    [1, 2, 3]
    """
    tmp = list1
    list1 = list2
    list2 = tmp
```

a) Explain why this code does not swap the contents of the two lists.

b) Modify the code so that it does swap the contents. Starter code for this problem can be found in `swap_contents.py`. To run the doctests on your implementation:

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

3. *Recursion.* Implement the `flatten()` function in Python. Given a nested list structure, it produces a new flattened list that contains all elements from the original structure. Use `isinstance(item, list)` to determine if `item` is a list.

   ```
   def flatten(item):
       """Produce a flattened version of the given structure.

       If item is a list, returns a new, flat list containing all
       the items contained within list or any of its elements. If
       item is not a list, returns a list containing item.

       >>> flatten(3)
       [3]
       >>> items = [1, 2, 3]
       >>> flattened = flatten(items)
       >>> flattened
       [1, 2, 3]
       >>> flattened is items  # verify flattened is a new list
       False
       >>> flatten([[], [1, 2], [[3, [4, 5]], 6], 7])
       [1, 2, 3, 4, 5, 6, 7]
       """
       # your code here
   ```

   Starter code for this problem can be found in `flatten.py`. To run the doctests on your implementation:

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

4. *Literals and operators.* Implement a `BitVector` type in C++17, representing a growable sequence of booleans. Your `BitVector` must support the following operations:

   - String literals that end with the _bv suffix construct a `BitVector` from the string with bits in order from left to right. A `0` character specifies `false` and a `1` character specifies `true`. Example:

     ```
     "1010"_bv  -->  [ true, false, true, false ]
     ```

     In other words, the literal (*external representation*) `"1010"_bv` gets converted to the data structure (*internal representation*) of a `BitVector` object that contains the elements `true, false, true, false`.
   - The `size()` member function returns the size of the `BitVector`. The return type should be `size_t`.
   - The `push_back()` member function takes in a `bool` and appends it to the end of the `BitVector`.
   - The `[]` operator returns the boolean at the given position. You may return by value or reference (i.e. you do not have to support modifying a `BitVector` using the `[]` operator).
   - The bitwise operators `&`, `|`, and `^`, when applied to two `BitVectors`, should result in a `BitVector` that consists of the `AND`, `OR`, and `XOR` of the two `BitVectors`, respectively. If they differ in size, then the operators should treat the smaller as if it were padded on the right with zeros.
   - The << stream insertion operator when applied to a `BitVector` should insert each boolean as a `0` or `1`. Example:

     ```
     cout << "1010"_bv;  -->  prints 1010
     ```

   Write your code in `BitVector.hpp`. We have provided a test case in `BitVector_test.cpp` and expected output in `BitVector_test.correct`.

   You may find this reference helpful.

   To compile and run tests, use the included Makefile:

   ```
   $ make bv
   ```