EECS 490 Final Review Exercises

True/False

1. Java allows a `double` to be converted to an `int` without an explicit cast.
2. An interpreter can apply rules from operational semantics to compute an expression or a statement.
3. A dispatch dictionary maps a message to the behavior or data represented by the message.
4. In Python, variables defined directly within the class (i.e. without `self`) are automatically private.
5. The default access level in Java for class members is public.
6. Java allows a class to implement multiple interfaces.
7. In C++, it is always an error if class `D` derives from both `B` and `C`, and both `B` and `C` derive from `A`.
8. In C++, runtime type information is used to determine whether or not a `dynamic_cast` should succeed.
9. In logic programming, every predicate has explicit input and output parameters.
10. A Makefile combines aspects of both declarative and imperative programming.
11. Template specialization allows the definition of a base case for a recursive template.
12. In C++, both class and function templates support partial specialization.
13. Variadic templates can be used to write function overloads that are type safe.
14. In the first phase of the MapReduce model, the individual computational nodes that apply the `map()` function must communicate and synchronize with each other.
15. A race condition is when multiple threads wait indefinitely for each other.

Free Response

1. Suppose we wanted to add a do/while statement to the simple language from our discussion on operational semantics:

$$S \rightarrow \textbf{do}\ S\ \textbf{while}\ B\ \textbf{end}$$

   a) Write a rule for the execution of the do/while statement in big-step operational semantics.

   b) Draw a derivation tree for **do** $x = (y * x)$ **while** $(x <= 3)$ **end**, with x having an initial value of 3 and y an initial value of 2.

2. Suppose we wanted to add sequencing to the language from our discussion on formal type systems:

$$E \rightarrow E\ ,\ E$$

   The value of such an expression is the same as the value of the second subexpression. Write a formal type rule for a sequencing expression.

3. Consider the following C++ code:

```
struct A {
  int x;
  virtual void foo();
};

struct B : A {
  int y;
  virtual void bar();
};

struct C : B {
  int z;
  virtual void baz();
};
```

Draw the layout of objects of type `A`, `B`, and `C`. Draw the vtables for each type.

4. Write a generic Java function to reverse the elements of a `Vector`, such that

```
Vector<String> vec = new Vector<String>();
vec.add("foo");
vec.add("bar");
vec.add("baz");
reverse(vec);
```

will result in `vec` containing `["baz", "bar", "foo"]`.

5. Write a Prolog predicate `all_even` that is true when its argument is a list that contains only even numbers. You may assume that the argument is a list, and that it only contains numbers:

```
?- all_even([1, 2, 3, 4]).
false.
?- all_even([2, 4]).
true.
?- all_even([]).
true.
```

6. Write a Prolog predicate `slice` that relates a list, an inclusive start index, an exclusive end index, and a second list containing the elements between those indices in the first list, preserving order:

```
?- slice([1, 2, 3, 4], 2, 2, X), !.
X = [].
?- slice([1, 2, 3, 4], 2, 4, X), !.
X = [3, 4].
```

7. Write a Prolog predicate `reverse` that relates a list to its reverse. You may **not** use `append` in your solution:

```
?- reverse([1, 2, a, b], X), !.
[b, a, 2, 1].
```

8. Write a Prolog predicate `all_reverse` that relates a list of lists to a second list that contains the reverse of each element of the first list, preserving order. You may use the `reverse` predicate:

```
?- all_reverse([[], [1], [2, 3], [a, b, c]], X), !.
X = [[], [1], [3, 2], [c, b, a]].
```

9. An efficient solution to count the number of ones in an integer is to divide it up into smaller pieces, such as 4-bit *nibbles*, and then look up the piece in a table to determine how many bits are in the piece. Write a Python function `gen_nibble_table()` that generates code for a nibble lookup table in C++. The end result should be something like the following:

```
int table[16] = {
  0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4
};
```

Do **not** hard-code this result. Instead, compute it and generate it programmatically.

10. Recall the `point` class template from lecture:

```
template <int N>
struct point {
  int coords[N];
  int &operator[](int i) {
    return coords[i];
  }
  const int &operator[](int i) const {
    return coords[i];
  }
};
```

Write a set of function overloads for `slice()` such that it takes a `point<N>` and an index k in $[0,N)$ and produces a `point<N-1>` that contains all but the kth coordinate from the original point. However, if `slice()` is called on a `point<1>`, it should return the original point unchanged. Examples (using the overloaded stream insertion operator from lecture):

```
cout << slice(point<3>{ 1, 2, 3 }, 1) << endl;
cout << slice(point<1>{ 3 }, 0) << endl;
```

This should print:

```
(1,3)
(3)
```