EECS 490 Final Review Exercises

True/False

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

Free Response

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

   $\frac{\langle s,\sigma\rangle \to \sigma_1 \quad \langle \text{while } b \text{ do } s \text{ end},\sigma_1\rangle \to \sigma_2}{\langle \text{do } s \text{ while } b \text{ end},\sigma\rangle \to \sigma_2}$

   $S \to$ **do** S **while** B **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:

   $\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1, t_2 : T_2}$

   $E \to E\ ,\ E$

   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.

   $\sigma(x) = 3\ ,\ \sigma(y) = 2$

3. Consider the following C++ code:

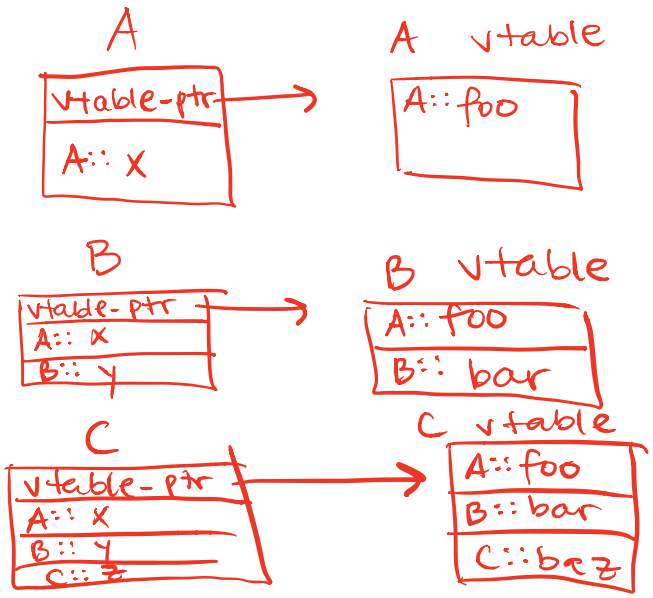
   $\frac{\langle y,\sigma\rangle \to 2 \quad \langle x,\sigma\rangle \to 3}{\langle (y * x),\sigma\rangle \to 6} \qquad \frac{\langle x,\sigma[x:=6]\rangle \to 6 \quad \langle 3,\sigma[x:=6]\rangle \to 3}{\langle (x <= 3),\sigma[x:=6]\rangle \to \text{false}}$

   $\frac{\langle x = (y * x),\sigma\rangle \to \sigma[x:=6] \qquad \langle \text{while } (x <= 3) \text{ do } x = (y * x) \text{ end},\sigma[x:=6]\rangle \to \sigma[x:=6]}{\langle \text{do } x = (y * x) \text{ while } (x <= 3) \text{ end},\sigma\rangle \to \sigma[x:=6]}$

```
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"]`.

```
static <T> void reverse (Vector<T> vec) {
  for (int i=0; i < v.size() / 2; i++) {
    T tmp = v.get(i);
    v.set(i, v.get(v.size()-1-i));
    v.set(v.size()-1-i, tmp);
  }
}
```

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.
```

```
all_even([]).
all_even([First|Rest]) :-
  First mod 2 =:= 0,
  all_even(Rest).
```

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].
```

```
slice(_, _, 0, []).
slice([First|Rest], 0, End, [First|Result]) :- NewEnd is End-1,
  slice(Rest, 0, NewEnd, Result).
slice([_|Rest], Start, End, Result) :- NewStart is Start-1, NewEnd is End-1,
  slice(Rest, NewStart, NewEnd, Result).
```

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].
```

```
reverse(List,Result) :- reverse_helper(List, [], Result).

reverse_helper([], SoFar, SoFar).
reverse_helper([First|Rest], SoFar, Result) :-
  reverse_helper(Rest, [First|SoFar], Result).
```

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]].
```

```
all_reverse([], []).
all_reverse([First|Rest], [RevFirst|RevRest]) :-
  reverse(First, RevFirst),
  all_reverse(Rest, RevRest).
```

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
};
```

```
def gen_nibble_table():
  table = 'int table[16] = {{ {} }};'
```

```
def count_ones(x):
  count = 0
```

```
counts = [ str(count_ones(i))
           for i in range(16)]
print(table.format(', '.join(counts)))
```

```
while x :
  if x % 2 == 1 :
    count += 1
  x = x//2
return count
```

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)
```

```
template <int N>
point<N-1> slice(const point<N> &p, int k) {
  point<N-1> result;
  for (int i=0; i<N; ++i) {
    if (i<k) {
      result[i] = p[i];
    } else if (i>k) {
      result[i-1] = p[i];
    }
  }
  return result;
}

point<1> slice(const point<1> &p, int k) {
  return p;
}
```