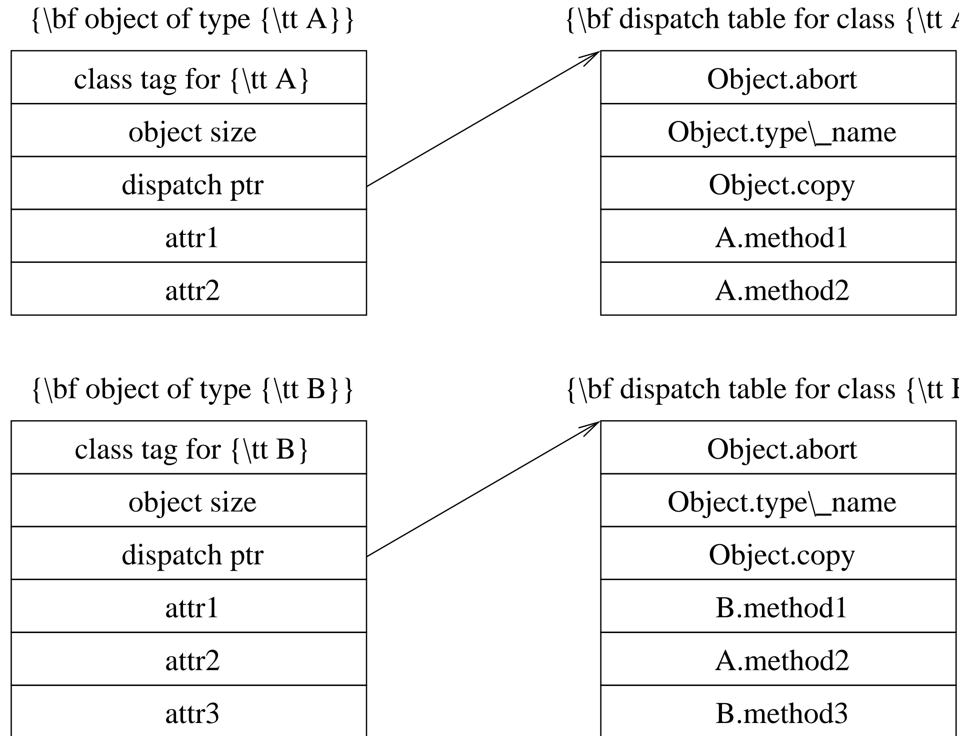
Solutions to Written Assignment 7

1. (a) The following diagram illustrates objects of type `A` and `B` along with their dispatch tables.



   (b) Note that we assume the existence of a label called `dispatch_error` that handles the case where `$a0` is null.

```
beq    $a0 $zero dispatch_error      # check for null obj
lw     $t0 8($a0)                    # load dispatch pointer
lw     $t0 16($t0)                   # load method2 ptr from table
jalr   $t0                           # jump to method
```

   (c) If `obj` has dynamic type `B`, then we correctly invoke `method2` on the object. All objects have a dispatch pointer at offset 8, so we correctly fetch the dispatch pointer. Furthermore, all classes that inherit from class `A` will have a pointer to the appropriate version of `method2` at offset 16 of their dispatch table. In this case, we will call the version of `method2` supplied by class `A`, since class `B` did not override it.

   Note that it is legal to pass an object of type `B` as the `self` object, since the layout of `A` is a prefix of the layout of `B`.

2. These rules treat arrays as objects with n attributes, all of type T. For convenience, arrays are indexed from 1 to n, rather than 0 to $n-1$.

$$\frac{\begin{array}{l} T = \begin{cases} X & \text{if } T_0 = \text{ SELF_TYPE and } so = X(\ldots) \\ T_0 & \text{otherwise} \end{cases} \\ so, S_1, E_1 \vdash e_1 : \text{Int}(n), S_2 \\ l_i = \text{newloc}(S_2) \text{ for } i = 0 \ldots n \text{ and each } l_i \text{ is distinct} \\ v_a = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ S_3 = S_2[v_a/l_0, D_T/l_1, \ldots, D_T/l_n] \\ E_2 = E_1[l_0/a] \\ so, S_3, E_2 \vdash e_2 : v_2, S_4 \end{array}}{so, S_1, E_1 \vdash \text{let } a : T_0[e_1] \text{ in } e_2 : v_2, S_4} \quad \text{[Array-Let]}$$

D_T is just the default value for objects of type T (e.g., 0 for `Int`s and `void` for most objects).

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \text{Int}(m), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ E(a) = l_a \\ S_2(l_a) = v_a \\ v_a = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ 1 \leq m \leq n \\ S4 = S3[v_2/l_m] \end{array}}{so, S_1, E \vdash a[e_1] \texttt{ <- } e_2 : v_2, S_4} \quad \text{[Array-Assign]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \text{Int}(m), S_2 \\ E(a) = l_a \\ S_2(l_a) = v_a \\ v_a = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ 1 \leq m \leq n \\ v = S_2(l_m) \end{array}}{so, S_1, E \vdash a[e] : v, S_2} \quad \text{[Array-Lookup]}$$

3. Here's one way to do it. This approach literally checks if this is the last time around the loop or not, and behaves accordingly. We also use the old false rule to handle loops where e_1 is never true.

$$\frac{so, S_1, E \vdash e_1 : \mathit{Bool}(\mathit{false}), S_2}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \mathit{void}, S_2} \quad \text{[Loop-False]}$$

$$\frac{\begin{array}{c} so, S_1, E \vdash e_1 : \mathit{Bool}(\mathit{true}), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash e_1 : \mathit{Bool}(\mathit{false}), S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_2, S_4} \quad \text{[Loop-True-Last]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \mathit{Bool}(\mathit{true}), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash e_1 : \mathit{Bool}(\mathit{true}), S_{peek} \\ so, S_3, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_3, S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_3, S_4} \quad \text{[Loop-True-Not-Last]}$$

Note that S_{peek} is just thrown away; we're just using it to make sure that this isn't the last time around the loop.

On a real machine, we can't usually throw away a store like this; this solution is a good example of operationaly semantics being more powerful than we can easily implement. An alternate approach is to

allocate a chunk of memory for the value of the loop expression, and write into it each time we traverse the loop.