1. [26 pts.]

   a. The result would be 8. The rule is saying that in an application of a closure to an argument, variable bindings in the closure’s environment take precedence over the binding of the value of the argument to the closure’s parameter. In other words, it’s like an inverse static scoping, in which outermost variable bindings take precedence over inner ones. (Of course this behavior would make no sense for a real language, it’s just to test understanding of operational semantics rules.) Note that the original rule (which describes what real Scheme would use) the function application would return 6, as the outer x and its value 4 are shadowed by the inner x and its value 3.

   b. i. $\lambda y.z \ y$
      
      ii. $\lambda z.y \ z$ (where z can be any variable other than y)
      
      iii. $\lambda y.y$

   c. In translation via erasure Java replaces type parameters with `Object`, since any type which the type parameter is instantiated with will be a subclass of `Object`. C++ does not have a class like `Object` which is the root of the entire inheritance hierarchy, without which this approach won’t work.

2. [24 pts.] The idea behind this problem, of creating and returning a function which captures some data which was in scope where the function was declared, was what most of Lecture #27 was devoted to discussing an example of. That example showed turning the transitions of a DFA into a function which mapped a state and symbol to a new state, and which captured the list of transitions in the function in which it was declared. Although this problem asks for the same idea to be used, it’s much simpler than the example discussed in lecture.

Here are three slightly different solutions:

```ocaml
let create_contact_list names =
  let rec search list name =
    match list with first::phone::rest ->
      if name = first
        then phone
        else search rest name
    in search names

let rec create_contact_list names =
  let rec lookup name =
    match names with first::phone::rest ->
      if first = name
        then phone
        else ((create_contact_list rest) name)
    in lookup
```

let rec create_contact_list names =
  fun name -> match names with
        first::phone::rest ->
          if first = name
          then phone
          else ((create_contact_list rest) item)

3. [10 pts.] 13 17

4. [25 pts.] One note before giving the solution: recall from CMSC 250 that zero is a multiple of 3 (the multiples of 3 are \{…, −9, −6, −3, 0, 3, 6, 9, …\}). Therefore \(\epsilon\) is not a valid string in this language. This is a language which is regular, but it’s atypical in that for this language it’s actually probably easier to just write the grammar directly, rather than writing the DFA and converting it to a regular grammar, which is usually simpler (although that could definitely work though, as shown below).

Your first thought might be to write a grammar which looked something like the following, which would generate strings where \(m + n\) is a multiple of 3 and then generate either one or two extra a’s or b’s, thereby forcing \(m + n\) to not be a multiple of 3:

\[
\begin{align*}
S & \rightarrow \text{aaa}S | \text{aa}Sb | aSbb | T \\
T & \rightarrow a | aa | b | bb | ab
\end{align*}
\]

The problem with these grammars are that they’re ambiguous. For example, in the grammar on the left:

\[
\begin{align*}
S & \rightarrow \text{aaa}S \\
& \rightarrow \text{aa}Sb \\
& \rightarrow aSbb \\
& \rightarrow T \\
T & \rightarrow a \\
& \rightarrow aa \\
& \rightarrow b \\
& \rightarrow bb \\
& \rightarrow ab
\end{align*}
\]

Here are several unambiguous versions:

\[
\begin{align*}
S & \rightarrow \text{aaa}S | T \\
T & \rightarrow aaTb | U \\
U & \rightarrow aUbb | V \\
V & \rightarrow Vbbb | W \\
W & \rightarrow a | aa | b | bb | ab
\end{align*}
\]

Below is a DFA which recognizes the language, and a grammar which is generated from it, using the construction shown in discussion section. Note that the DFA keeps track of the number of a’s generated and whether they’re a multiple of 3, and once a b is seen it’s able to keep track of whether the remainder of the string consists of only b’s and would preserve the total string length as being not a multiple of 3:
Note: the productions for Y would be optional (the grammar would generate the same language if they weren’t shown).

Any completely-correct answer should have the following properties:

p1: [9 pts.] (completeness) It generates every valid string.
p2: [8 pts.] (correctness) It generates only valid strings (it does not generate any invalid strings).
p3: [8 pts.] (ambiguity) It’s unambiguous.

5. [15 pts.] This is almost the identical example which was written on the board and discussed on Friday before the exam, and again on Monday two days before the exam.

- a. f1
  f1
  f2
  g1
  g2
  g2
- b. Only z.