Here are the “questions” from the code file:

Q0: Notice how the algorithm for append is the usual one but where lists are nonempty and the message-send syntax is a little different than function calls.
Append only checks whether the tail of the list exists instead of checking if the list is empty, since that is the base case (since this is a non-empty list).

When using the message-sending syntax, the programmer has to decide which object to send a message to in order to generate a new object or cause mutations.

Note the function body could also be
```
(send (if tail (send tail append other) other) cons hd)
```

Q1: For ne-word-list%, why is there a let* *outside* the class expression?
This caches and shares the large result of an expensive computation and effectively acts as a private static field shared across all instances of ne-word-list%. If it were inside the class expression, this computation would occur on every instantiation.

Notice how lexical scope and first-class classes combine to let the concept of a private static field “emerge” with no special support – it’s just an idiom.

Q2: Why does ne-word-list% have to override cons and map, i.e., how is the behavior different than if it inherited them?
This is necessary to provide the guarantee that the resulting object is actually a word list. For example, if we do not override cons, the superclass method will be used and the new-head won’t be checked for being a word.

Q3: Why does ne-word-list% have to use (= (send this length 1)) in various places?
Calling get-tail on an empty tail is an error, so it’s only possible to safely check that the tail is nonempty by checking the length. The tail itself cannot be accessed directly by the subclass because it is private.

Q4: Why does (fail-app2b) fail while none of the others do?
Append eventually does `(send other cons head)`, and example2 (other) is ne-word-list whose cons expects elements to be words. However, example1 contains a non-word.

Q5: Why must the definition of extra-words come before the call to (super-new) in ne-extensible-word-list%?
When super-new is called, the superclass will run the `check-word` *defined in ne-extensible-word-list%* (by overriding) to see if the head is a word in the dictionary. If `extra-words` is not defined, that method call will fail. For this reason, the private fields necessary for a subclass must be defined before the super-class calls methods in the subclass that need these fields.

Q6: Why must the dynamic type-check for the tail come after the call to (super-new) in ne-extensible-word-list%?

When performing the dynamic type-check for the tail, since the object's tail field has not been initialized, the superclass length method will fail. In general, an object needs to be properly instantiated from its superclass before you start sending messages to it.

Q7: For the use of define/augment

A: Why doesn't define/override work?

define/pubment only allows subclasses to augment.

B: Why doesn't super work?

Use of super is prohibited inside define/augment and if it were allowed, it would cause an infinite loop.

C: How could you allow a nested inner in subclasses if you wanted that?

You would add a call to the subclass's check-words with a default argument like `(inner #f check-words w)`.

- If augment does not call inner, subclass augment does nothing (use augride to allow subclasses to override an augmented function)
- If augment calls inner, the subclass' augment can then indeed augment