Below is the Pig Latin code provided in lab.

```
(define (pigl wd)
  (if (pl-done? wd)
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))

(define (pl-done? wd)
  (vowel? (first wd)))

(define (vowel? letter)
  (member? letter '(a e i o u)))
```

**Q1: Is the same for all versions!**

Q1: We LOVE helper procedures and think that you should too! But to test your understanding of how these helper procedures are working, please re-write the bolded code in pigl without calling the helper procedures pl-done? and vowel? Without changing the behavior of the function pigl, (pl-done? wd) can be replaced with:

```
(member? (first wd) '(a e i o u))
```

---

**Grading (out of 1 point):**

- Invalid Scheme (-1 point)
- Using vowel? (-1 point)
- Switching arguments of member? (-0.5 point)
- Forgetting to call first (-0.5 point)

**Q3: Is the same for all versions!**

Q3: Write the procedure multiply that multiplies all of the numbers in a sentence as shown by the example calls below.

```
STk> (multiply '(1 2))
2
STk> (multiply '(10 3 2))
60
STk> (multiply '())
1
```

```
(define (multiply sent)
  (if (empty? sent)
      1
      (* (first sent) (multiply (bf sent)))))
```

---

**Grading (out of 2 points):**

- return () as the base-case (we want to work with numbers! And return a number!) (-0.5 points)
  This was REALLY common!
- using sentence as a combiner (we want to work with numbers! And return a number!) (-0.5 points)
  This was REALLY common!
- small mistake (-0.5 points)
- Proper start of definition "(define (multiply sent)" and proper condition (no more than -1.5 off)
- Leaving out the base case/recursive call (-1 point each)
- Using list operations instead of sentence operations (-0.5 point)
- Syntax of cond/if is incorrect (-0.5 point)
- Three really small errors (-1 point)
Version 1

Q2: Fill in the blank to show what scheme would print.
STk> (define (a b c)
   (if (= b 1)
       c
       (+ c (a (- b 1) c))))
a
STk> (a 4 7)
28 (1 point)

Q4: How many times is * called in the following code: (1 point)

STk> (define (square x) (* x x))
STk> (define (weird x y) (* y y y y))
STk> (weird (square (* 1 1)) (* 3 3))

Using applicative order: ____4_____
Using normal order: ____5_____

Version 2

Q2: Fill in the blank to show what scheme would print.
STk> (define (a b c)
   (if (= b 1)
       c
       (+ c (a (- b 1) c))))
a
STk> (a 4 3)
12 (1 point)

Q4: How many times is * called in the following code: (1 point)

STk> (define (square x) (* x x))
STk> (define (weird x y) (* y y y y))
STk> (weird (square (* 1 1)) (* 3 3))

Using applicative order: ____4_____
Using normal order: ____6_____

Version 3

Q2: Fill in the blank to show what scheme would print.
STk> (define (a b c)
   (if (= b 1)
       c
       (+ c (a (- b 1) c))))
a
STk> (a 4 6)
24 (1 point)

Q4: How many times is * called in the following code: (1 point)

STk> (define (square x) (* x x))
STk> (define (weird x y) (* y y y y))
STk> (weird (square (* 1 1)) (* 3 3))

Using applicative order: ____4_____
Using normal order: ____5_____
Q2: Fill in the blank to show what scheme would print.

STk> (define (a b c)
    (if (= b 1)
        c
        (+ c (a (- b 1) c)))

a
STk> (a 4 5)

20 (1 point)

Q4: How many times is * called in the following code: (1 point)

STk> (define (square x) (* x x))
STk> (define (weird x y) (* y y y))
STk> (weird (square (* 1 1)) (* 3 3))

Using applicative order: ______4______

Using normal order: ______ 4____