A.A. Functions have a uniform and simple pattern of evaluation: All of the function's arguments are evaluated, before the function is applied to them. Scheme's `let` does not follow this pattern, because it leaves some of its arguments — namely, the local variables and the sublists that they head — unevaluated. Therefore `let` cannot be a function. It is a keyword.

A.B. The given `let` expression is equivalent to

```
((lambda (var1 var2) body) val1 val2)
```

Another correct answer is the curried version

```
(((lambda (var1) (lambda (var2) body)) val1) val2)
```

B. Here is my solution:

```
;Input f: A function of one argument.
;Input lazy: A lazy list.
;Return: A lazy list. The results of applying f to the elements of lazy.
(define map-lazy
  (lambda (f lazy)
    (if (null? lazy)
        '()
        (cons (f (car lazy))
              (lambda () (map-lazy f ((cdr lazy))))))))
```

C.A. In the given code, the variable `myArray` is being allocated on the stack. But the stack is not allowed to hold such large variables. When I start using `myArray`, I access regions of memory illegally, and the operating system terminates the program with a seg fault.

To allocate such a large variable, we need to allocate it on the heap, even though its use is local to this function.

C.B. Here is my C code:

```
void myFunc() {
    int *myArray = malloc(1000000000 * sizeof(int));
    assert(myArray != NULL);
    /* ...do stuff with myArray here... */
    free(myArray);
}
```

D.A. Here are the three syntax error fixes in `main`:

```
attachLeg(&dog);
printf("%d\n", dog.numLegs);
setEyePairs(&spider, v[1]);
```

D.B. Here is the printed output:

```
5
9 2 1 1 6
9 2 1 1 6
```

E.A. With static scoping, the parent of a function application's local frame is the frame in which the function was defined/constructed. With dynamic scoping, the parent of a function application's local frame is the frame in which the function is being applied/called/invoked.

E.B. This `f` is behaving as if it's statically scoped, because it implicitly uses `x = 3`, then `x = 5`, then `x = 5`. It's definitely not dynamically scoped, because it doesn't pick up the `x = 6` from the calling frame in the third example. (By the way, the language in question is Mathematica.)

E.C. This `g` implicitly uses `x = 3`, then `x = 3`, then `x = 3`. It seems that the value of `x`, that was in effect when `g` was defined, was permanently "baked into" `g`. That's not static scope or dynamic scope. It's something else. It's some kind of partial evaluation upon definition.