3.2:

The local variable allocation differences in these languages is due to the fact that each language stores its variables in memory differently. Fortran 77 stores its variables statically so that there will be enough room in memory to allow for other local variables to allocate during runtime. Languages like Algol store local variables on the stack so that they can use more variables and functions that are allocated on the stack. Lisp also stores local variables on the stack.

Java Example:

```
public class staticVariables {
     public static void main(String[] args) {
          int x = 5;
          decrement(x);
          System.out.println(x);
     }

     public static void decrement(int n) {
          n--;
     }
}
```

This code would not work if local variables were allocated statically as opposed to on the stack because the value of 'n' would only exist for the method call of decrement. The program would always return 5.

Scheme Example:

```
(define (factorial x)
     (let ((result 1))
          (if (= n 0)
               result
               (* n (factorial (- n 1))))))
(display (factorial 4))
```

This code would not work because it would not remember the value of result within the recursive calls to factorial.

3.4:

Nested loop example:

```
public static void main(String[] args) {
     int x = 10;
     for (int i = 0; i < 5; i++) {
          int y= 20;
          for (int j = 0; j < 2; j++) {
               // code that uses x and y
          }
          // j is not in scope, but y is live
     }
     // i and y are not in scope, but x is still live
}
```

Try-catch block example:

```
public static void main(String[] args) {
     int x = 10;
     try {
          int y = 20;
     } catch (Exception e) {
          System.out.println(x);
          // y is not in scope, but is still live
     }
}
```

Lambda expression example:

```
public static void main(String[] args) {
     int x = 10;
```

```
    Runnable r = () -> {
        int y = 20;
        // some code that uses x and y
    };
    // y is not in scope, but is still live
}
```

3.5:

C:

Line 2: a=1

Line 3: b=2

Line 4: middle() function is defined

Line 5: b=a

Line 6: inner() function is defined

Line 7: print a,b is called, prints 1 1

Line 8: return to reference environment of middle()

Line 9: a=3

Line 11: print a,b prints ‘3 2’ because the block that includes b at line 5 has ended

Line 14: print a,b prints ‘1 2’ because the blocks at line 5 and 8 have ended

Modula-3:

Line 2: a=1

Line 3: b=2

Line 4: middle() function is defined

Line 5: b = a

Line 6: inner() function is defined

Line 7: print a,b prints 1 1

Line 8: inner() returns but b = a is still true because middle() has not returned

Line 9: a = 3

Line 11: print a,b will print ‘3 1’ because reference for a is at line 8 and b at line 5

Line 12: middle() returns, so a = 1 and b = 2

Line 14: print a,b prints ‘1 2’ because middle() returned and a = 1 and b = 2

3.7:

a) Brad’s code has a memory leak because he never calls delete_list to free the nodes that he allocated to L.
b) The list T must also all be freed, not just L

3.14:

The program prints 1 1 2 2 using static and dynamic scoping. Variable x is always in scope, so its position in memory is irrelevant. x is kept in scope for the whole program using static scoping. Using dynamic scoping, x is always in scope.

3.18:

If the language uses shallow binding, the program will print 1 1 3 3. The variables are found in the environment where the function is defined, rather than where it is called.
With deep binding, the program prints 1 0 3 0. The variables are found in the environment where the function is called, as opposed to where it is defined like shallow binding.