6.1

These two statements are not contradictory. The compiler can optimize the code when it detects patterns in commutative arithmetic operations in order to reduce the number of calculations that need to be performed. The example given in the book shows how the compiler can rearrange the order of evaluation to match a previous computation, and then substitute a variable in place of one of the calculations. Clearly, this will result in the same answer, but with one fewer calculation than the version which has not been optimized.

6.23

While loop example:

```
line = read_line();

while(!all_blanks(line))
{
     consume_line(line);

     line = read_line();
}
```

6.24

There is a good alternative that does not rely on a goto statement, the following code could be C, C++, Java, etc...

```
int i, j, allZero;

for(i = 0; i < n; i++)
{
     allZero = 1;
     for(j = 0; j < n; j++)
     {
          if(A[i][j] != 0)
          {
               allZero = 0;
               break;
          }
     }

     if(allZero)
     {
          break;
     }
}
```

7.1

For most situations, structural equivalence is a bad thing. It is quite possible that a programmer will create two different records, which have the same data type structure. In this case, the compiler would recognize them as being of the same type, when really they are not. In order to avoid this situation, a programmer may have to create filler variables in order to avoid having one record be recognized as a different record simply because the structure of the two is the same.

7.2

Under structural equivalence, all would be recognized as being the same. Under strict name equivalence, A and B would be recognized as the same. Under loose name equivalence, A, B, and C would all be recognized as being the same.

7.8

The array would consume 20 bytes of memory. The compiler will have to position the values that come after each char on an even number. Since the char is only 1 byte, a byte will have to be skipped in order to reach that even number. Because of this, only 18 bytes will actually be used, and 2 will be wasted.

8.3

```c
#include <stdio.h>
#include <stdlib.h>

int eval1();
int eval2();
int eval3();
void eval(int a, int b, int c);

int main(int argc, char** argv)
{
    eval(eval1(), eval2(), eval3());

    return 0;
}

int eval1()
{
    printf("Function eval1 is running.\n");

    return 1;
}

int eval2()
{
    printf("Function eval2 is running.\n");

    return 2;
}

int eval3()
{
    printf("Function eval3 is running.\n");

    return 3;
}


void eval(int a, int b, int c)
{
    printf("The parameter values are %d, %d, %d.\n", a, b, c);
}
```

When I ran this test on Fedora using gcc, the functions as parameters were evaluated right to left.

8.4

One explanation for Unix systems using the gcc compiler, is that a variable that is unitialized by the programmer, is automatically initialized to 0 by the compiler, and also made static. The behavior on other systems may be different because they do not use gcc. If that is the reason, then the behavior would definitely be a result of the compiler implementation.

8.6

```c
#include <stdio.h>
#include <stdlib.h>

void paramEval(int val, int ref, int name, int result);

int main(int argc, char** argv)
{
    int* param = (int*)malloc(sizeof(int));

    paramEval(*param, &param, param, (*param = 10));

    free(param);

    return 0;
}

void paramEval(int val, int ref, int name, int result)
{
    printf("Pass by value: %d\n", val);
    printf("Pass by reference: %d\n", ref);
    printf("Pass by name: %d\n", name);
    printf("Pass by result: %d\n", result);
}
```

8.9

The example given in the book has the value being passed by reference. Because of that, it is able to be overwritten. Obviously that is not desirable behavior because as the example shows, this allows the overwriting of constant values. In order to get around this, newer versions of Fortran would either need to pass by value, or not allow constants to be passed as parameters. The former of these two options is the more likely scenario.