1. Consider the following two C program files:

```c
/* file1.c */
#include <stdio.h>
extern int x;
extern int foo( int y );
static int a = 420; float x = 4.20;

int main( int argc, char *argv[] ) {
    int i = x;
    for ( i = 0; i < 4; ++i )
        (void) printf( "%d", foo( i ) );
    return 0;
}

/* file2.c */
#include <stdio.h>
extern int a;
float x = 4.20;

void foo( int z ) {
    static int b = 15;
    ++b;
    (void) printf( "%d", z );
    (void) printf( "%d", a );
}
```

Trying to separately compile each file and then link the resulting object modules:

```bash
gcc -c file1.c  file1.c –> cpp –> ccomp –> as –> file1.o

gcc –c file2.c  file2.c –> cpp –> ccomp –> as –> file2.o

gcc file1.o file2.o file1.o & file2.o –> ld –> a.out/.exe
```

results in just one error being reported. We discussed some of the problems/complications imposed on the compiler to be able to perform static semantic type checking with separate compilation.

What error will be reported (specify the symbol name and a general description of what the problem is). **Hint:** The error will be reported in the 3rd gcc call which attempts to link the already compiled and assembled object modules. **Hint Hint:** Think scope.

Assuming we fixed this error so the program will fully compile/link. How many times does the variable b in function foo() get initialized?

Can we change the initialization of b in file2.c to be 

```c
static int b = z;
```

Why or why not?

Identify two other potential semantic errors in this program that the C compiler and linker did not detect, but lint will identify.

1) 

2)
2. Consider the following pseudocode:

```
TYPE Cell = RECORD
  VAR info : INTEGER;
  VAR next : POINTER TO Cell;
END;
TYPE Link = POINTER TO Cell;
TYPE PTC  = Link;
VAR first : Link;
VAR last  : Link;
VAR a     : PTC;
VAR b     : POINTER TO Link;
VAR c, d  : POINTER TO Cell;
VAR e     : POINTER TO RECORD
  VAR info : INTEGER;
  VAR next : Link;
END
VAR f     : POINTER TO POINTER TO Cell;
```

Which variables are considered equivalent under strict name equivalence?

<table>
<thead>
<tr>
<th>group 1</th>
<th>group 2 (opt)</th>
<th>group 3 (opt)</th>
<th>group 4 (opt)</th>
</tr>
</thead>
</table>

Which variables are considered equivalent under loose name equivalence?

<table>
<thead>
<tr>
<th>group 1</th>
<th>group 2 (opt)</th>
<th>group 3 (opt)</th>
<th>group 4 (opt)</th>
</tr>
</thead>
</table>

Which variables are considered equivalent under structural equivalence?

<table>
<thead>
<tr>
<th>group 1</th>
<th>group 2 (opt)</th>
<th>group 3 (opt)</th>
<th>group 4 (opt)</th>
</tr>
</thead>
</table>

The C compiler uses ________________ equivalence for all types except ________________
for which the C compiler uses ________________ equivalence.

Given the following ANSI/ISO C variable definitions, identify which expressions will produce a static semantic compiler error. Hint: Think modifiable l-value.

- A) No compiler error
- B) Compiler error

```
int i = 5;
float f = 1.5;
int *iPtr = &i;
float *fPtr = &f;
iPtr = (int *) fPtr;  ____  *iPtr = (int) *fPtr;  ____
++fPtr;  ____  (float *) iPtr = fPtr;  ____
fPtr = &(i + f);  ____  ++( (float *) iPtr );  ____
i = **&iPtr;  ____  i = *&iPtr;  ____
```