CSC-363: Problem Set 3: Language Characteristics III

1. The Collatz sequence may be finite or (possibly) infinite based on the input value. We generate iterative values according to the following function.

$$f(n) = \begin{cases} \frac{n}{2} & n \,\%\, 2 = 0 \\ 3n + 1 & n \,\%\, 2 = 1 \end{cases}$$

For example, we generate the Collatz sequence with initial input 17 as follows:

$$\begin{gathered} f(17) = 52 \\ f(52) = 26 \\ f(26) = 13 \\ f(13) = 40 \\ f(40) = 20 \\ f(20) = 10 \\ f(10) = 5 \\ f(5) = 16 \\ f(16) = 8 \\ f(8) = 4 \\ f(4) = 2 \\ f(2) = 1 \end{gathered}$$

Implement a function using suspension that will generate a Collatz sequence (whether finite or infinite).

2. Many main-stream programming languages, including Java and C#, have begun to add constructs that support concurrent (parallel) execution. Java's thread mechanism is a good example of this. Another proposed concurrent execution mechanism is method level parallelism. Ordinary method calls are synchronous. This means that the caller of a method suspends execution while the called method executes, and the caller then resumes after the called method returns.

a) Methods that are procedures (i.e., return void) are called for side-effects since they return no explicit value to the caller. This suggests it may be possible for the caller of a procedure to resume execution immediately, and run while the called procedure is executing. This execution mechanism is what we mean by method-level parallelism; the calling method and the called method may execute in parallel.
Under what circumstances is it safe to use method-level parallelism? That is, under what circumstances will concurrent execution of the two methods give the same result as ordinary synchronous execution?

b) Of course many method calls are to functions not procedures. For these calls, the caller expects to receive a return value. Is method-level parallelism of any value for function calls? That is, does it ever make sense to resume execution of the caller even though the method just called has yet to return its result value?

3. Structural equivalence in languages that contain structs (like C, C++ and C#) is defined as follows.

> Struct `S1` is structurally equivalent to struct `S2` if `S1` and `S2` contain the same number of fields and corresponding fields (in order of declaration) in `S1` and `S2` are structurally equivalent. (The names of corresponding fields within the two structs need not be the same.)
>
> Two pointers are structurally equivalent if the types they point to are structurally equivalent. That is, `S1*` (a pointer to type `S1`) is structurally equivalent to `S2*` if and only if `S1` is structurally equivalent to `S2`.
>
> Two arrays are structurally equivalent if they have the same size and their component types are structurally equivalent.
>
> Each scalar type is structurally equivalent only to itself.

(a) Assume a C-like language in which structs may contain fields declared to be scalars (int, float, etc.), arrays, pointers, and (nested) structs. Give pseudocode for an algorithm that decides if two structs, `S1` and `S2`, are structurally equivalent.

(b) Most languages, including C, C++ and C#, state that the order in which fields are declared is unimportant. That is, rearranging field declarations in a struct has no effect other than possibly changing the size of the struct.

Given this observation, it might make sense to change the rule for structural equivalence of structs so that two structs are structurally equivalent if they contain the same number of fields and it is possible to reorder the fields of one struct so that corresponding fields are structurally equivalent after reordering. That is, the order of fields in a struct no longer matters (nor does the name of fields). Thus the following two structs are now considered structurally equivalent:

```
struct S                            struct T
{                                   {
    int   f1;                           float g1;
    float f2;                           int   g2;
}                                   }
```

Update your algorithm from part (a) to implement this revised definition of structural equivalence. Illustrate your algorithm on the following set of structs. Is `S1` structurally equivalent to `S2`? Why?

```
struct S1                           struct S3
{                                   {
    S1* f1;                             S4* h1;
    S2* f2;                             int h2;
    S3* f3;                         }
}
                                    struct S4
struct S2                           {
{                                       S3* j1;
   S4* g1;                              int j2;
   S1* g2;                          }
   S2* g3;
}
```