Assignments are due on the due date before class and have to include this cover page. Plagiarism in assignment answers will not be tolerated. By submitting their answers to this assignment, the authors named above declare that its content is their original work and that they did not use any sources for its preparation other than the class notes, the textbook, and ones explicitly acknowledged in the answers. Any suspected act of plagiarism will be reported to the Faculty's Academic Integrity Officer and possibly to the Senate Discipline Committee. The penalty for academic dishonesty may range from failing the course to expulsion from the university, in accordance with Dalhousie University’s regulations regarding academic integrity.
Question 1 (15 marks)

(a) (5 marks) Whether two types are structurally equivalent is ultimately a matter of interpretation: the language designer decides which differences between two types are significant and which ones are not. On page 304 of the textbook, the following algorithm for determining structural equivalence of two types A and B is given:¹ We convert the two types A and B into two strings $s_A$ and $s_B$ such that the two types are equivalent if and only if the two strings are the same. String $s_A$ is defined inductively as follows (the definition of string $s_B$ is analogous): If A is a built-in type, $s_A$ is the name of type A. If A is a composite of a sequence of subtypes $A_1, A_2, \ldots, A_k$, then $s_A$ consists of the name of the type constructor used to construct A, followed by the strings $s_{A_1}, s_{A_2}, \ldots, s_{A_k}$. For example, if A is defined as

```c
struct A {
    char x;
    int *y;
};
```

the string $s_A$ is "struct char pointer int".

Argue why you cannot use this definition of type equivalence to decide whether the following two types A and B are equivalent:

```c
struct A {
    char x;
    struct B *y;
};
```

```c
struct B {
    char x;
    struct A *y;
};
```

(b) (10 marks) Here is an alternate definition of type equivalence. Every composite type provides element selectors as mechanisms to access the parts it is composed of. For example, the type B defined in part (a) provides element selectors .x and .y to access its constituent parts; an array type provides [i] to access its elements. By applying element selectors recursively, we can define valid sequences of element selectors. For an object of type B in part (a), for example, the sequences .y.y.y.y and y.y.y.x are valid and lead to objects of type B and char, respectively, while the sequence .y.y.x.y is not valid because .y.y.x leads to an object of type char, which does not have a part accessible through the selector .y. Now we define that two types A and B are equivalent if they have the same (possibly infinite) set of valid sequences of element selectors and each such sequence that leads to a built-in type from A leads to the same built-in type from B. It is not hard to see that, based on this definition, the two types A and B in part (a) are structurally equivalent.

Describe an algorithm for testing whether two types are structurally equivalent using this definition. Argue that your algorithm is correct, that is, that it classifies two types as structurally equivalent if and only if they are according to the definition just given. (Hint: Recall the algorithm for minimizing a DFA. It should serve as a good starting point.)

¹This is not a verbatim copy of the algorithm given in the textbook but a more rigorous inductive description that captures the same notion of structural type equivalence.