Computation Exercises 2: State

1. Consider the small-step operational semantics of the language *While*. Write down all of the evaluation steps of the program $(z := x; x := y); y := z$, with the initial state $s = (x \mapsto 5, y \mapsto 7)$. Give the full derivation tree for the first step in this evaluation.

2. Consider the small-step operational semantics of the language *While*. Write down all of the evaluation steps of the program

$$\texttt{while}\ x < 4\ \texttt{do}\ x := x + 2$$

   given the initial state $s = (x \mapsto 1)$. Give the full derivation trees for the first four steps.

3. Consider adding the increment expression $x{+}{+}$ to the language *While*. The expression $x{+}{+}$ returns the value of the variable x and then updates the value of x to be one greater than the old value; its semantics is given by the following rule:

$$(\text{W-EXP.PP})\ \frac{}{\langle x{+}{+}, s\rangle \rightarrow_e \langle n, s[x \mapsto n']\rangle}\ s(x) = n, n' = n \underline{+} 1$$

   (Note that the ++ operator is only applied to variables — we do not allow $E{+}{+}$ for general expressions E.)

   (a) Give the full execution path for the program $x := (x{+}{+}) + (x{+}{+})$ from the initial state $(x \mapsto 2)$.

   (b) Give an operational semantics rule for $++x$, which increments x and *then* returns the result.

4. Consider what happens if we add a 'side-effecting expression' of the form

$$\texttt{do}\ C\ \texttt{return}\ E$$

   to the language *While*. The idea here is that the above expression first runs the command C and then returns the value of E. For example,

$$\texttt{do}\ x := 1\ \texttt{return}\ x$$

   sets x to 1 and returns 1. Extend the small-step operational semantics of *While* to include this kind of expression.

5. Consider the *While* language extended with parallel composition of commands: $C \parallel C$. The semantics of parallel composition is given by *interleaving* the execution steps of the two composed commands in an arbitrary fashion. This behaviour is expressed formally by the axioms and rules:

$$\frac{\langle C_1, s\rangle \rightarrow_c \langle C_1', s'\rangle}{\langle C_1 \parallel C_2, s\rangle \rightarrow_c \langle C_1' \parallel C_2, s'\rangle} \qquad \frac{\langle C_2, s\rangle \rightarrow_c \langle C_2', s'\rangle}{\langle C_1 \parallel C_2, s\rangle \rightarrow_c \langle C_1 \parallel C_2', s'\rangle}$$

$$\frac{}{\langle \texttt{skip} \parallel \texttt{skip}, s\rangle \rightarrow_c \langle \texttt{skip}, s\rangle}$$

(a) Consider the command $(x := 1) \parallel (x := 2; x := (x + 2))$, run with initial state $s = (x \mapsto 0)$. How many possible final values for x does this command have? Write down at least one evaluation path for each of these different values.

(b) How many different evaluation paths exist for obtaining the final value 4? Explain why this is so.

(c) A useful operation in concurrency is atomic compare-and-swap. This operation is added to the *While* language in the form of a new boolean expression $\texttt{CAS}(x, E, E)$. To execute the operation $\texttt{CAS}(x, E_1, E_2)$, first E_1 and then E_2 are evaluated to numbers n_1 and n_2 in the usual way. Then, *in a single step,* the operation compares the value of variable x with number n_1; if the values are equal, it updates the value of x to be number n_2 and returns `true`; otherwise, it simply returns `false`.

Extend the operational semantics with rules for `CAS` that implement this behaviour.

6. Suppose that $\langle C_1; C_2, s\rangle \rightarrow_c^* \langle C_2, s'\rangle$. Show that it is not necessarily the case that $\langle C_1, s\rangle \rightarrow_c^* \langle \texttt{skip}, s'\rangle$. (Note: it will always be the case that $\langle C_1, s\rangle \rightarrow_c^* \langle \texttt{skip}, s''\rangle$ for some s''.)