Computation Answers 2: State

Aside: What is the aim of these exercises?

These first two exercise sheets have introduced you to operational semantics – both big- and small-step. One good way to think about the purpose of these operational semantics is to place them into the context of designing and implementing a programming language. A designer might conjure a new programming language out of their imagination. They then write down the semantics of that language (perhaps using big- or small-step semantics) and hand this document to a team of programmers. The programmers would then implement the language – perhaps with a compiler of the sort you are writing in the WACC lab.

Of course, in practice, many languages are not specified as formally and precisely as our *While* language. For example, the standard that defines the JavaScript programming language is a 566-page document full of English-language text and pseudocode (`http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`). Writing a completely formal semantics for a large programming language involves a lot of work, and many designers of popular languages have chosen to make do with a less formal specification, which does take (much) less time and effort to write.

In future exercises, you will make use of the operational semantics we have for the *While* language to prove important properties about the language. For example, if I run the same program twice, do I always get the same answer? Imagine how hard testing your programs can be if this property didn't hold. Without a formal semantics, we might have a strong intuition that a property like this might hold, but there would be no way to *prove* it. Once we begin to think about properties such as memory safety, which are important for security (Does your language permit buffer-overflows?), we require proper proofs and intuition is no longer sufficient. It is for reasons like this that formal semantics are now being developed, even for large, complex, existing languages like JavaScript (`http://jscert.org`).

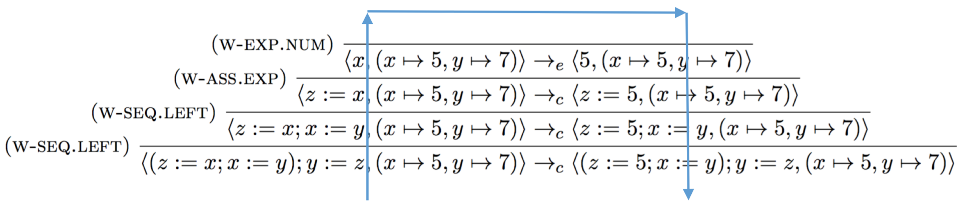
1. Consider the small-step operational semantics of the language *While*. Write down all of the evaluation steps of the program $(z := x; x := y); y := z$, with the initial state $s = (x \mapsto 5, y \mapsto 7)$. Give the full derivation tree for the first step in this evaluation.

Solution. This question required of you to provide the evaluation steps of a particular *While* program, starting from a particular state. Perhaps you wrote an interpreter for the language, ran this program in your interpreter, and would now like to make sure that your interpreter is adhering to the semantics.

We will show the derivation of the first step of the evaluation first.

$$
\text{(W-SEQ.LEFT)}\ \dfrac{\text{(W-SEQ.LEFT)}\ \dfrac{\text{(W-ASS.EXP)}\ \dfrac{\text{(W-EXP.NUM)}\ \dfrac{}{\langle x, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_e \langle 5, (x \mapsto 5, y \mapsto 7)\rangle}}{\langle z := x, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_c \langle z := 5, (x \mapsto 5, y \mapsto 7)\rangle}}{\langle z := x; x := y, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_c \langle z := 5; x := y, (x \mapsto 5, y \mapsto 7)\rangle}}{\langle (z := x; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_c \langle (z := 5; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle}
$$

We will use this derivation to illustrate the steps necessary to build a full derivation in small-step semantics. This we can very loosely express in words as "bottom-left-top-left-top-right-bottom-right" and visually as in Figure 1, on the next page. We start from $\langle (z := x; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle$. What does it mean for this command to do



$$
(\text{W-SEQ.LEFT})\ \dfrac{(\text{W-SEQ.LEFT})\ \dfrac{(\text{W-ASS.EXP})\ \dfrac{(\text{W-EXP.NUM})\ \dfrac{}{\langle x, (x \mapsto 5, y \mapsto 7)\rangle \to_e \langle 5, (x \mapsto 5, y \mapsto 7)\rangle}}{\langle z := x, (x \mapsto 5, y \mapsto 7)\rangle \to_c \langle z := 5, (x \mapsto 5, y \mapsto 7)\rangle}}{\langle z := x; x := y, (x \mapsto 5, y \mapsto 7)\rangle \to_c \langle z := 5; x := y, (x \mapsto 5, y \mapsto 7)\rangle}}{\langle (z := x; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle \to_c \langle (z := 5; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle}
$$

Figure 1: Direction of the derivation for small-step semantics

one step? Since the command is a sequence, it means that we the leftmost command of the sequence, which is $z := x; x := y$, needs to do one step. The question then becomes: what does it mean for $\langle z := x; x := y, (x \mapsto 5, y \mapsto 7)\rangle$ to do one step? This is also a sequence, therefore it means that the leftmost command of the sequence, which is $z := x$, needs to do one step. Now, what does it mean for $\langle z := x; (x \mapsto 5, y \mapsto 7)\rangle$ to do one step? This is an assignment whose operand (x) is not fully evaluated—therefore, it means that $\langle x, (x \mapsto 5, y \mapsto 7)\rangle$ needs to do one step, and the only step possible, given our rules, is $\langle x, (x \mapsto 5, y \mapsto 7)\rangle \to_e \langle 5, (x \mapsto 5, y \mapsto 7)\rangle$. At this point, we have reached the top left part of the derivation, and have moved to the top right part. What is now left is to propagate the obtained information downwards, which is usually completely straightforward, but requires a bit of focussed attention. In this case, this amounts to substituting x for 5 three times, as shown in the derivation.

All evaluation steps of the derivation are:

$$
\begin{aligned}
&\langle (z := x; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle \\
&\to_c \langle (z := 5; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle \\
&\to_c \langle (\texttt{skip}; x := y); y := z, (x \mapsto 5, y \mapsto 7, z \mapsto 5)\rangle \\
&\to_c \langle x := y; y := z, (x \mapsto 5, y \mapsto 7, z \mapsto 5)\rangle \\
&\to_c \langle x := 7; y := z, (x \mapsto 5, y \mapsto 7, z \mapsto 5)\rangle \\
&\to_c \langle \texttt{skip}; y := z, (x \mapsto 7, y \mapsto 7, z \mapsto 5)\rangle \\
&\to_c \langle y := z, (x \mapsto 7, y \mapsto 7, z \mapsto 5)\rangle \\
&\to_c \langle y := 5, (x \mapsto 7, y \mapsto 7, z \mapsto 5)\rangle \\
&\to_c \langle \texttt{skip}, (x \mapsto 7, y \mapsto 5, z \mapsto 5)\rangle
\end{aligned}
$$

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.

Solution. Let $P = \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2$. The derivations for the first four steps are:

$$(\text{W-WHILE})\ \frac{}{\langle \texttt{while}\ x < 4\ \texttt{do}\ x := x+2, (x \mapsto 1)\rangle \rightarrow_c \langle \texttt{if}\ x < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 1)\rangle}$$

$$(\text{W-COND.BEXP})\ \frac{(\text{W-BEXP.LEFT})\ \dfrac{(\text{W-EXP.NUM})\ \dfrac{}{\langle x, (x \mapsto 1)\rangle \rightarrow_e \langle 1, (x \mapsto 1)\rangle}}{\langle x < 4, (x \mapsto 1)\rangle \rightarrow_b \langle 1 < 4, (x \mapsto 1)\rangle}}{\langle \texttt{if}\ x < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 1)\rangle \rightarrow_c \langle \texttt{if}\ 1 < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 1)\rangle}$$

$$(\text{W-COND.BEXP})\ \frac{(\text{W-BEXP.LT})\ \dfrac{}{\langle 1 < 4, (x \mapsto 1)\rangle \rightarrow_b \langle \texttt{true}, (x \mapsto 1)\rangle}\ 1 < 4}{\langle \texttt{if}\ 1 < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 1)\rangle \rightarrow_c \langle \texttt{if}\ \texttt{true}\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 1)\rangle}$$

$$(\text{W-COND.TRUE})\ \frac{}{\langle \texttt{if}\ \texttt{true}\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 1)\rangle \rightarrow_c \langle x := x+2; P, (x \mapsto 1)\rangle}$$

The remaining steps are

$$\begin{aligned}
&\langle x := x+2; P, (x \mapsto 1)\rangle\\
&\rightarrow_c \langle x := 1+2; P, (x \mapsto 1)\rangle\\
&\rightarrow_c \langle x := 3; P, (x \mapsto 1)\rangle\\
&\rightarrow_c \langle \texttt{skip}; P, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle \texttt{while}\ x < 4\ \texttt{do}\ x := x+2, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle \texttt{if}\ x < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle \texttt{if}\ 3 < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle \texttt{if}\ \texttt{true}\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle x := x+2; P, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle x := 3+2; P, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle x := 5; P, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle \texttt{skip}; P, (x \mapsto 5)\rangle\\
&\rightarrow_c \langle \texttt{while}\ x < 4\ \texttt{do}\ x := x+2, (x \mapsto 5)\rangle\\
&\rightarrow_c \langle \texttt{if}\ x < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 5)\rangle\\
&\rightarrow_c \langle \texttt{if}\ 5 < 4\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 5)\rangle\\
&\rightarrow_c \langle \texttt{if}\ \texttt{false}\ \texttt{then}\ x := x+2; P\ \texttt{else}\ \texttt{skip}, (x \mapsto 5)\rangle\\
&\rightarrow_c \langle \texttt{skip}, (x \mapsto 5)\rangle
\end{aligned}$$

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.

Solution. (a)

$$\begin{aligned}
&\langle x := (x{+}{+}) + (x{+}{+}), (x \mapsto 2)\rangle \\
&\rightarrow_c \langle x := 2 + (x{+}{+}), (x \mapsto 3)\rangle \\
&\rightarrow_c \langle x := 2 + 3, (x \mapsto 4)\rangle \\
&\rightarrow_c \langle x := 5, (x \mapsto 4)\rangle \\
&\rightarrow_c \langle \texttt{skip}, (x \mapsto 5)\rangle
\end{aligned}$$

Note that in Java and JavaScript, the ++ operator has this kind of semantics. In C, the side-effects of ++ can occur in different places so $x = x{+}{+}$ may give x its original value. (The C language specification is loose, so $x = x{+}{+}$ may instead increment x, depending on the implementation.)

(b) For this part of this question, you are asked to play the role of the language designer, and write down the operational rule for a new feature for the *While* language. Unfortunately we weren't able to give you free reign to make up any feature you like, because then we would have had no way to give you a helpful model answer. So, we asked you to write an operational rule for the $++x$ feature, which had been described in English first. Here is a good small-step rule for $++x$:

$$(\textsc{w-exp.ppl})\ \frac{}{\langle ++x, s\rangle \rightarrow_e \langle n', s[x \mapsto n']\rangle}\ s(x) = n, n' = n \underline{+} 1$$

If you want to add other features to the *While* language, go right ahead! You could even post them on Piazza, where we can chat about what sorts of programs they might allow the users of your language to write.

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.

Solution. Again, you're in the role of the language designer, writing down rules for a new feature.

$$\frac{\langle C, s\rangle \rightarrow_c \langle C', s'\rangle}{\langle \texttt{do}\ C\ \texttt{return}\ E, s\rangle \rightarrow_e \langle \texttt{do}\ C'\ \texttt{return}\ E, s'\rangle}$$

$$\frac{}{\langle \texttt{do skip return}\ E, s\rangle \rightarrow_e \langle E, s\rangle}$$

Notice in particular the way the first rule propagates the new state s', which records any side effects which the command C may have had.

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.

Solution. We promised last week that some of the differences between big-step and small-step operational semantics would become apparent when you considered concurrency. That is a part of the point of this question. We can define interleaving semantics for concurrency with relative ease using small-step semantics. Consider how hard it is to do so with big-step.

(a) The execution path to get the answer 1 is:

$$\begin{aligned}
&\langle x := 1 \parallel (x := 2; x := x + 2), (x \mapsto 0)\rangle\\
&\rightarrow_c \langle x := 1 \parallel (\texttt{skip}; x := x + 2), (x \mapsto 2)\rangle\\
&\rightarrow_c \langle x := 1 \parallel x := x + 2, (x \mapsto 2)\rangle\\
&\rightarrow_c \langle x := 1 \parallel x := 2 + 2, (x \mapsto 2)\rangle\\
&\rightarrow_c \langle x := 1 \parallel x := 4, (x \mapsto 2)\rangle\\
&\rightarrow_c \langle x := 1 \parallel \texttt{skip}, (x \mapsto 4)\rangle\\
&\rightarrow_c \langle \texttt{skip} \parallel \texttt{skip}, (x \mapsto 1)\rangle\\
&\rightarrow_c \langle \texttt{skip}, (x \mapsto 1)\rangle
\end{aligned}$$

One execution path to get the answer 3 is:

$$\begin{aligned}
&\langle x := 1 \parallel (x := 2; x := x + 2), (x \mapsto 0)\rangle\\
&\rightarrow_c \langle x := 1 \parallel (\texttt{skip}; x := x + 2), (x \mapsto 2)\rangle\\
&\rightarrow_c \langle x := 1 \parallel x := x + 2, (x \mapsto 2)\rangle\\
&\rightarrow_c \langle \texttt{skip} \parallel x := x + 2, (x \mapsto 1)\rangle\\
&\rightarrow_c \langle \texttt{skip} \parallel x := 1 + 2, (x \mapsto 1)\rangle\\
&\rightarrow_c \langle \texttt{skip} \parallel x := 3, (x \mapsto 1)\rangle\\
&\rightarrow_c \langle \texttt{skip} \parallel \texttt{skip}, (x \mapsto 3)\rangle\\
&\rightarrow_c \langle \texttt{skip}, (x \mapsto 3)\rangle
\end{aligned}$$

There is another: you could bring the execution of $x := 1$ forward by one step. The execution paths that give the answer 4 are:

$$\begin{aligned}
&\langle x := 1 \parallel (x := 2; x := x + 2), (x \mapsto 0)\rangle\\
&\to_c \langle \texttt{skip} \parallel (x := 2; x := x + 2), (x \mapsto 1)\rangle\\
&\to_c \langle \texttt{skip} \parallel (\texttt{skip}; x := x + 2), (x \mapsto 2)\rangle\\
&\to_c \langle \texttt{skip} \parallel x := x + 2, (x \mapsto 2)\rangle\\
&\to_c \langle \texttt{skip} \parallel x := 2 + 2, (x \mapsto 2)\rangle\\
&\to_c \langle \texttt{skip} \parallel x := 4, (x \mapsto 2)\rangle\\
&\to_c \langle \texttt{skip} \parallel \texttt{skip}, (x \mapsto 4)\rangle\\
&\to_c \langle \texttt{skip}, (x \mapsto 4)\rangle
\end{aligned}$$

$$\begin{aligned}
&\langle x := 1 \parallel (x := 2; x := x + 2), (x \mapsto 0)\rangle\\
&\to_c \langle x := 1 \parallel (\texttt{skip}; x := x + 2), (x \mapsto 2)\rangle\\
&\to_c \langle x := 1 \parallel x := x + 2, (x \mapsto 2)\rangle\\
&\to_c \langle x := 1 \parallel x := 2 + 2, (x \mapsto 2)\rangle\\
&\to_c \langle \texttt{skip} \parallel x := 2 + 2, (x \mapsto 1)\rangle\\
&\to_c \langle \texttt{skip} \parallel x := 4, (x \mapsto 1)\rangle\\
&\to_c \langle \texttt{skip} \parallel \texttt{skip}, (x \mapsto 4)\rangle\\
&\to_c \langle \texttt{skip}, (x \mapsto 4)\rangle
\end{aligned}$$

$$\begin{aligned}
&\langle x := 1 \parallel (x := 2; x := x + 2), (x \mapsto 0)\rangle\\
&\to_c \langle x := 1 \parallel (\texttt{skip}; x := x + 2), (x \mapsto 2)\rangle\\
&\to_c \langle x := 1 \parallel x := x + 2, (x \mapsto 2)\rangle\\
&\to_c \langle x := 1 \parallel x := 2 + 2, (x \mapsto 2)\rangle\\
&\to_c \langle x := 1 \parallel x := 4, (x \mapsto 2)\rangle\\
&\to_c \langle \texttt{skip} \parallel x := 4, (x \mapsto 1)\rangle\\
&\to_c \langle \texttt{skip} \parallel \texttt{skip}, (x \mapsto 4)\rangle\\
&\to_c \langle \texttt{skip}, (x \mapsto 4)\rangle
\end{aligned}$$

(b) There are three such evaluation paths, as shown above. The command $x := 1$ can occur before $x := 2$. It can also occur after the value $[x \mapsto 2]$ has been read, or after the expression $2 + 2$ has been evaluated to 4. The point is that the command $x := x + 2$ is not atomic, because of the one-step expression evaluation.

(c)

$$\frac{\langle E_1, s\rangle \to_e \langle E_1', s'\rangle}{\langle \texttt{CAS}(x, E_1, E_2), s\rangle \to_b \langle \texttt{CAS}(x, E_1', E_2), s'\rangle}$$

$$\frac{\langle E_2, s\rangle \to_e \langle E_2', s'\rangle}{\langle \texttt{CAS}(x, n_1, E_2), s\rangle \to_b \langle \texttt{CAS}(x, n_1, E_2'), s'\rangle}$$

$$\frac{}{\langle \texttt{CAS}(x, n_1, n_2), s\rangle \to_b \langle \texttt{true}, s[x \mapsto n_2]\rangle}\ s(x) = n_1$$

$$\frac{}{\langle \texttt{CAS}(x, n_1, n_2), s\rangle \to_b \langle \texttt{false}, s\rangle}\ s(x) \neq n_1$$

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

Solution. Suppose that $\langle C_1, s\rangle \to_c^* \langle \texttt{skip}, s''\rangle$ for some $s'' \neq s'$. Then in order for $\langle C_1; C_2, s\rangle \to_c^* \langle C_2, s'\rangle$, it is enough that $\langle C_2, s''\rangle \to_c^* \langle C_2, s'\rangle$. That is, we need to choose C_2 so that it eventually reduces to itself in a different state — it loops!

Let $C_1 = \texttt{skip}$, $C_2 = \texttt{while true do } x := 1$, $s = (x \mapsto 0)$ and $s' = (x \mapsto 1)$. We have

$$\begin{aligned}
&\langle \texttt{skip}; \texttt{while true do } x := 1, s\rangle \\
&\to_c \langle \texttt{while true do } x := 1, s\rangle \\
&\to_c \langle \texttt{if true then } x := 1; C_2 \texttt{ else skip}, s\rangle \\
&\to_c \langle x := 1; C_2, s\rangle \\
&\to_c \langle \texttt{skip}; C_2, s'\rangle \\
&\to_c \langle C_2, s'\rangle
\end{aligned}$$

That is, $\langle C_1; C_2, s\rangle \to_c^* \langle C_2, s'\rangle$. Yet it is not the case that

$$\langle \texttt{skip}, (x \mapsto 0)\rangle \to_c^* \langle \texttt{skip}, (x \mapsto 1)\rangle.$$