1. (Operational semantics)

(a) Figure 1 shows a simpler version of the language of commands that we saw in class. In this language, there are no booleans, only integers, as in C. As such, expressions like $n_1 > n_2$ will evaluate to 1 when true, and 0 when false. Moreover, conditionals will branch to the true branch when the expression in the guard is non-zero and to the false case otherwise. The same idea applies to the while loops. Define big-step and small-step operational rules for this modified semantics. For those constructs in the language that have the same semantics in the lecture notes, just say so (you don’t need to write them twice).

(b) Using your semantics, show derivations that establish the following as legal evaluations:
$$(1 \times 2) + (3 \times x) \rightarrow^* 8$$ and likewise $$((1 \times 2) + (3 \times x), \sigma) \rightarrow 8$$ where $\sigma(x) = 2$.
In the first case, you are writing a sequence of derivations of small-step reductions, and in the second case you are writing one big-step derivation. Note that the syntax $\sigma[x \mapsto n]$ defines a new map $\sigma'$ such that $\sigma'(x) = n$ and $\sigma'(y) = \sigma(y)$ for all $y \neq x$.

(c) Give a big-step derivation for the following:
$$(\text{while } x \text{ do } (x := x + (-1)), \sigma) \rightarrow \sigma'$$ where $\sigma(x) = 1$ and $\sigma' = \sigma[x \mapsto 0]$.

---

variables $x, y, z \in V$
integers $n, m \in \mathbb{Z}$
expressions $e ::= x | n | e_1 + e_2 | e_1 \times e_2 | e_1 = e_2 | e_1 < e_2 | !e_1$
commands $c, d ::= \text{skip} | x := e | \text{if not0 } e \text{ then } c_1 \text{ else } c_2 | \text{while not0 } e \text{ do } c | c_1 ; c_2$

Figure 1: Extended language of commands
2. (Abstract interpretation) Consider this program, which computes factorial:

```c
// input is integer y
x := 1;
while (y != 0) do
    x := x * y;
    y := y - 1;
done
// x contains y!
```

We wish to use abstract interpretation to prove that, given a positive integer $y$ as input, this function always produces a positive integer as output.

(a) Suppose we use as our abstract domain the rule of signs, so that numbers are represented as one of $A = \{+, -, 0, \perp, \top\}$ (ordered as shown in the lecture notes). Further suppose we have $S$ such that $S(x) = \perp$ and $S(y) = +$. Call the program above $c_{\text{fact}}$. Then we want to prove that $(S, c_{\text{fact}}) \rightarrow S'$ where $S'(x) = +$.

Using the operational rules given in the lecture notes, we will end up computing a fixed-point by repeated application of rules for while (slide 28). Perform this computation presenting the result as per the control-flow graph visualization on slide 33. That is, show what $x$ and $y$ would be mapped to on each iteration until you reach a fixpoint for the loop, and then (according to the top rule on slide 28) return the final result $S'$. Is it the case that $S'(x) = +$? If not, what is $S'(x)$? What went wrong (i.e., why did we get this answer)? Hint: try using the abstract interpretation implementation we provided to check your answer.

(b) Construct a new abstract domain $A'$ for which we can show that $S'(x)$ is positive given positive input. Draw a picture showing the lattice structure for $A'$, and describe (briefly) operations in your new domain (if they are obvious, you can just say that you use the obvious abstract operations). Finally, compute a fixpoint approximation of $c_{\text{fact}}$ in your new abstract domain showing the desired result.