Exercise 1F-2. Language Design [5 points].
One aspect from Hoare's *Hints On Programming Language Design* that relates to my programming experience is the concept of simplicity. Hoare states that some programming language designs have prioritized modularity to improve simplicity, allowing a programmer to work with a language even with limited understanding. My experience with high-level languages such as Python have reflected this. Python's simple and abstracted syntax encourages clarity and ease of use. However, due to its simplistic syntax, there have been cases where I have had to dealt with unexpected errors or behavior. As a simple example, the "+" operator in Python can be used in several cases. Numerical values can be summed with it, but types such as strings can also be concatenated with it. If I wanted to add the strings "5" and "6" numerically, I would have to first cast them to integers. Doing so without casting would result in "56", a result that could be overlooked if not careful. In short, I agree with Hoare in that significant modularity can be overwhelming in certain situations.

A different aspect from Hoare's *Hints On Programming Language Design* that I disagree with is his view on comment conventions, specifically his criticism of special bracketing symbols. In my opinion, although it is true that omission or misplacement of a comment bracket can result in issues, this concern is miniscule in comparison to the convenience that special bracketing symbols provide. The symbols provide the ability to insert large comments when needed, and any errors that arise from it can typically be easily caught by the developer. In short, rather than introducing awkward problems, I believe special bracketing comment conventions actually introduce a significant amount of convenience.

Exercise 1F-3. Simple Operational Semantics [3 points].
In order to extend the Aexp sub-language with a division operator, we can introduce a new rule of inference:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad n_2 \neq 0}{\langle e_1 / e_2, \sigma\rangle \Downarrow n_1 / n_2}$$

To be more precise, we evaluate the expressions e_1 and e_2 to n_1 and n_2, respectively. We can express division as the value n_1/n_2. As an additional requirement, we must ensure n_2 is not 0 to avoid division by 0.

Exercise 1F-4. Language Feature Design, Large Step [10 points].
In order to extend the natural-style operational semantics judgment $\langle c, \sigma\rangle \Downarrow \sigma'$ with one new rule for dealing with the `let`, we can introduce a new rule of inference:

$$\frac{\langle e, \sigma\rangle \Downarrow n \quad \langle c, \sigma[x := n]\rangle \Downarrow \sigma'}{\langle \text{let } x = e \text{ in } c, \sigma\rangle \Downarrow \sigma}$$

We first evaluate the expression e to n in our current state. Then, we evaluate the command c in a modified state that maps x to the new value n, which results in a new state.

Questions assigned to the following page: 4 and 5

Since x only exists in the scope of the `let` statement, we return the original state σ after evaluating c within the modified state.

Exercise 1F-5. Language Feature Design, Small Step [10 points].
In order to extend the set of redexes to accommodate `let`, we need to include the following to the set of redexes:

$$\text{let } x = e \text{ in } c$$

Continuing, in order to extend the set of contexts, we can add a new context for `let`. When doing so, we want to make sure to evaluate the expression e first:

$$H ::= \ldots \mid \text{let } x = H \text{ in } c$$

Finally, in order to extend the reduction rules, we can add two new rules. First, we need to evaluate the expression e into a value n:

$$\langle \text{let } x = e \text{ in } c, \sigma \rangle \to \langle \text{let } x = n \text{ in } c, \sigma \rangle$$

Continuing, once we have evaluated e, we can bind x locally to n and and simplify:

$$\langle \text{let } x = n \text{ in } c, \sigma \rangle \to \langle c, \sigma[x := n] \rangle$$