Exercise 1F-2. Language Design [5 points]. In *Hints On Programming Language Design*, Hoare makes several bold statements about how programming languages should be designed. One of these is that a programming language should "encourage and assist the programmer to write clear self-documenting code, and even perhaps to develop and display a pleasant style of writing." In my experience, most of the code you write in industry is building off of code written by other people. Oftentimes, you are debugging or adding features to a massive code base that existed long before you started working on it. In such cases, it is extremely helpful if the original programmers left documentation or easily readable code for you to understand what a program is doing. Otherwise, you spend a lot more time trying to decipher their program rather than writing new code. Thus, I wholeheartedly agree that a well designed programming language would encourage programmers write clean and readable documentation and code.

Hoare also says that the efficiency of object code was not considered important because the speed and capacity of computers was increasing rapidly. This was true when he wrote his paper in 1973. However, he disagreed, saying that the efficiency loss still cannot be justified. Today, with Moore's law coming to an end, the speed and capacity of computers are no longer growing as rapidly as they were back then. Thus, Hoare is even more correct now because programming language designers can no longer rely on improvements in hardware to offset inefficiencies in object code.

One of Hoare's claims that I disagree with is that the modularity of programming languages should not replace simplicity. Modularity refers to being able to work with a language with a limited understanding of only part of it. I believe that modularity provides a lower barrier of entry to learning a language and greatly simplifies it. For example, the C++ Standard Library provides many useful data structures such as queues, stacks, dictionaries, etc. C++ programmers can use these data structures without understanding the underlying implementation, which allows them to write working and efficient code with less time spent learning the language. Removing this modularity would mean the programmer would either have to implement these data structures themselves, which has a steep learning curve, or using a simpler but less efficient solution.

Exercise 1F-3. Simple Operational Semantics [3 points]. If we extend the Aexp sub-language for IMP with a division operator, we need to add rules of inference for the new operator. This is shown below:

$$\frac{<e_1, \sigma> \Downarrow n_1 \qquad <e_2, \sigma> \Downarrow n_2}{<e_1 / e_2, \sigma> \Downarrow \lfloor n_1 / n_2 \rfloor}$$

Note: Since this is an integer operation, this rule performs division such that the result is rounded down to the nearest integer, hence the floor operator. Also, if attempting to divide by zero, this operation is undefined.

Exercise 1F-4. Language Feature Design, Large Step [10 points]. The let command can be defined by the following inference rule:

$$\frac{\dfrac{<e,\sigma> \Downarrow n}{<x{:=}e,\sigma> \Downarrow \sigma'[x{:=}n]} \qquad <c,\sigma'> \Downarrow \sigma''}{<\text{let } x{=}e \text{ in } c,\sigma> \quad \Downarrow \quad \sigma''[x{:=}\sigma(x)]}$$

Exercise 1F-5. Language Feature Design, Small Step [10 points]. The redexes, contexts, and local reduction rules for the let command are:

$$<\text{let } x{=}e \text{ in } c,\sigma> \rightarrow <x{:=}e; c; x{:=}\sigma(x),\sigma>$$

$$H ::= <x{:=}H_1; c; x{:=}\sigma(x),\sigma>,\ e{=}H_1[r]$$

1 HW1 (select all pages: your first page has your name and bookkeeping, and all others are anonymous))

- **0 pts** Correct