Exercise 1F-2. Language Design [5 points]. Comment on some aspect from Hoare's *Hints On Programming Language Design* that relates to your programming experience. Provide additional evidence in favor of one his points and against one of his points. Do not exceed three paragraphs. Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter. Readers should be able to identify your main claim, the arguments you are bringing to bear, and your conclusion.

One point that really resonated with me in this paper was the argument made for the Readability of code. A few years ago, I had was working on a project during an internship that involved developing software to perform statistical analysis of some data. For this task, there were two programming languages that were considered: R and Python. R seems like the obvious choice since it was developed specifically to do statistical analysis. And while Python definitely has statistical tools and libraries, it's much more . In the end, however, we decided to use Python for the sake of readability. The majority of the team had no experience with R but were very familiar with Python. To make sure other members would be able to look at the code later and easily make out what the program was doing, Python was the obvious choice. This also lends support to another point brought up in the paper: Debugging. If other members need to be able to easily update and debug on the code in future then it should be in a language they're comfortable with. While this situation wasn't a case of "this language is better designed than this other language", these principles that drove the design of the languages still played an important part in our decision.

Also in the paper, Hoare seems to go on a vindictive rant against reference variables. He even says that they are "...a step backward from which we may never recover." His objection to reference variables stems from many different issues: little control over what part of memory can get modified, confusion in syntax, performance overhead, etc. Hoare paints reference variables as this irredeemable sin in programming (at the very least he doesn't really discuss to merits to having pointers or references part of the language's design). This isn't something I agree with. I think, when used right, having pointers in your language allows for some truly powerful and useful control from the programmer (dynamic memory, polymorphism, etc.) I do agree with Hoare that pointers and references do open the door for some nasty behavior, but I'd say that's more of a worry for the programmer than it is indicative of bad language design.

Exercise 1F-3. Simple Operational Semantics [3 points]. Consider the IMP language discussed in class, with the Aexp sub-language extended with a division operator. Explain what changes must be made to the operational semantics (big-step only). Write out formally any new rules of inference you introduce.

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

The big issue with division is dividing by zero. The operation can only carry out if the denominator (in this case, e_2) does not evaluate to 0. The other thing to define is the division operation itself. Specifically here, this is the integer division operation. This naturally brings up the issue of dealing with remainders in the division. For integer division, any remainders will get thrown out. However, this means that there will be a loss of information that we have to accept (e.g. the operation 99/100 would evaluate to the value 0, despite 1 being a more acceptable answer).

Exercise 1F-4. Language Feature Design, Large Step [10 points]. Extend the natural-style operational semantics judgment $\langle c, \sigma\rangle \Downarrow \sigma'$ with one new rule for dealing with the `let`

and to changes to other variables.

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

For this new rule, we give three hypotheses for the `let` command, that need to be done in order. The first step is to evaluate $x := e$, which will produce a new state, $\sigma[x := n]$. This new state is then used to carry out the command, c, which will produce another new state σ'. Finally, we need to restore the original value of x while maintaining any other changes to the other variables, so we restore the value of x in σ' by carrying out $x := \sigma(x)$, which produces the final state $\sigma'[x := \sigma(x)]$

Exercise 1F-5. Language Feature Design, Small Step [10 points]. Extend the set of redexes, contexts and reduction rules for the contextual-style operational semantics that we discussed in class to account for the `let` command introduced above.

$H ::= ...|let\ x = H\ in\ c$

$r ::= ...|let\ x = n\ in\ c$

$\langle let\ x = n\ in\ c, \sigma\rangle \rightarrow \langle c; x := \sigma(x), \sigma[x := n]\rangle$

Like was stated in the previous problem, the `let` command boils down to three steps. First, the expression e must be evaluated to a value n. Once that is done, the command can be reduced, the reduction step being evaluating $x := n$ and then adding a new command $x := \sigma(x)$ to be in a sequence after c (where $\sigma(x)$ is the original value of x before the `let` command began).

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

- **0 pts** Correct