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.

Perhaps my favorite idea from Hoare's piece lies in his quote "If a programming language is regarded as a tool to aid the programmer, it should give [them] the greatest assistance in the most difficult aspects of [their] art". This is ultimately the basis for Hoare's succeeding claims, and I believe it to still hold to this day. I appreciate Hoare referring to the program's design as an "art", and interestingly enough that perception of Computer Science is what drew me to it several years ago. I'd argue that writing a program is no different than translating a poem from English to German. The programmer is merely an intermediary between arbitrary human thought and machine-interpretable commands.

At the advent of powerful transformer-based LLMs, this importance is greater than ever before. Hoare's concept of code being more readable than it is writeable, not only had set foundational programming conventions that are now leveraged in the development and training of such LLMs to enable them to assist in program synthesis but is also truer than ever in that if we would like an LLM to aid us in the process of writing a program, the feasibility of writing the program is far less of an importance than us as humans to understand what exactly that LLM had generated. In essence, today we as humans are *reading* a far greater fraction of code that we *did not write*, than ever before, and this concept of readability >> writeability is more relevant than ever. In this sense, Hoare's ideas have withstood the test of time.

It should be noted however, that Hoare's assertions pertaining to keeping a programming language both safe and simplistic undermine the real-world complexity of such an implementation. We can consider Rust as an example of a programming language highly regarded as being "safe" due to its ownership and borrowing system. However, in no way is Rust considered a "simplistic" language to learn, compared to say Python or Java, languages frequently used in introductory CS courses. There is an unavoidable tradeoff between the safety of a programming language, and its safety. These are only two dimensions nitpicked from Hoare's paper, but this tradeoff spans multiple dimensions, and must be considered in the very conceptualization of a programming language, with the target audience, target domain, and target purpose all kept in mind. I believe Hoare undermines this crucial principle, and could at the very least acknowledge and elaborate on it far more.

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.

Let us introduce the following inference rule for Aexp to account for this new division operation:

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

Exercise 1F-4. Language Feature Design, Large Step [10 points]. Consider the IMP language with a new command construct "`let` $x = e$ `in` c". The informal semantics of this construct is that the Aexp e is evaluated and then a new local variable x is created with lexical scope c and initialized with the result of evaluating e. Then the command c is evaluated. We also extend IMP with a new command "`print` e" which evaluates the Aexp e and "displays the result" in some un-modeled manner but is otherwise similar to `skip`.

We expect (the curly braces are syntactic sugar):

```
x := 1 ;
y := 2 ;
{ let x = 3 in
  print x ;
  print y ;
  x := 4 ;
  y := 5
} ;
print x ;
print y
```

to display "3 2 1 5".

Extend the natural-style operational semantics judgment $\langle c, \sigma\rangle \Downarrow \sigma'$ with one new rule for dealing with the `let` command. Pay careful attention to the scope of the newly declared variable and to changes to other variables.

Let us extend the natural language style operational semantics judgment $\langle c, \sigma \Downarrow \sigma'\rangle$ with a new let command using the following rule, defined as follows:

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

Where recover$(x,\ \sigma',\ \sigma)$ results in the state identical to σ', except x in this new state is replaced with the value associated with the variable x if x is defined in σ, but otherwise leaves the variable x undefined if x is not defined in σ. i.e. It handles the shadowing and scope of variable 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.

We can extend the set of redexes to account for the `let` with the following:
$\mid$ `let` $x = n$ `in` c

We can extend the set of contexts to account for the `let` with the following:
$\mid$ `let` $x = H$ `in` c

We can extend the set of reduction rules to account for the `let` with the following:
$\mid \langle$ `let` $x = n$ `in` $c,\ \sigma\rangle \rightarrow \langle\ c,\ \sigma[x := n]_{scoped}\ \rangle$

Where $\sigma[x := n]_{scoped}$ simply means that after the respective command c finishes executing, x is restored to its original value (or lack thereof, i.e. undefined) as it was in the state σ. i.e. It accounts for shadowing and the scoped nature of the variable x defined by `let`.