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.

Answer:
In his paper *Hints On Programming Language Design*, Hoare outlines what he considers essential properties of programming languages. The concept that I most agree with is that the burden of program safety - and specifically memory safety - falls on the programming language itself, not the programmer. Conversely, the point that I disagree with is that program writeablity is a relatively unimportant concern when designing a programming language.

The point I most agree with is that "The objective of security has also been widely ignored; it is believed instead that coding errors should be removed by the programmer," (page 7) Specifically, with anything that relates to memory management in C++, it is up to the programmer to manage all memory, to make sure not to dereference a null pointer, to make sure that pointers actually point to valid data, and to free up that memory exactly when it is no longer needed. Even in programming languages that boast garbage collectors, the compilers cannot identify if a null value is being used until runtime. It is my belief that programming languages should, at their core, address these concerns so that the programmer can spend more time developing functionality rather than having to fix memory or pointer problems. In fact, the language Rust aims to do just that:"Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time." (https://www.rust-lang.org/) Rust's syntax enables ensures memory safety, and programming languages should attempt to move towards such model. Overall, memory management and program safety should be a concern of the programming language and not the programmer.

One point I disagree with Hoare on is his claim that "The readability of programs is immeasurably more important that their writeability." (page 3) This claim omits the fact that a lot of programming is tinkering, trying things a certain way, and then another, repeatedly. If program writeability is overlooked then using a language based on this principle might be unnecessarily tedious. This is supported by my experience of preferring Python over C++ for the simple reason that it is much easier to attempt something small quickly in Python. Overall, I disagree with Hoare's claim that program writeablity is unimportant, as tinkering is an essential part of my programming journey.

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.

Answer:
We extend Aexp with:

$$\begin{array}{rcll} e & ::= & ... & \\ & | & e_1/e_2 & \text{for } e_1, e_2 \in \text{Aexp} \end{array}$$

We need to change the operational semantics in two ways: the first is that we need to add new judgment(s) to deal with this rule, and exceptions to deal with the special case of division by zero. Let us add these two judgments:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1/e_2, \sigma\rangle \Downarrow \texttt{DivisonByZeroError}}\ \mathsf{div-zero} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1/e_2, \sigma\rangle \Downarrow n_1/n_2}\ \mathsf{div}$$

div may only be used if $n_2 \neq 0$.

Peer Review ID: 63152768 — enter this when you fill out your peer evaluation via gradescope

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.

Answer:
The new rule for dealing with the `let` command is:

$$\frac{\langle e, \sigma \rangle \Downarrow n \qquad \langle c, \sigma[x := n] \rangle \Downarrow \sigma'}{\langle \texttt{let } x = e \texttt{ in } c, \sigma \rangle \Downarrow \sigma'[x := \sigma(x)]} \; \mathsf{let-com}$$

This command says that the expression e evaluates to n, at which point n is bound to x in σ. We then evaluate c with this modified σ and it yields a new state σ'. We then revert x to its value in the original σ as it has gone out of scope.

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.

Answer:
We can expand the set of redexes to:

$$\begin{array}{rcl} r & ::= & \ldots \\ & | & \texttt{let}\ x = n\ \texttt{in}\ c \end{array}$$

We can then expand the set of contexts to include the `let` command:

$$\begin{array}{rcl} H & ::= & \ldots \\ & | & \texttt{let}\ x = H\ \texttt{in}\ c \end{array}$$

We can finally expand the set of reduction rules to include this new `let` redex with:

$$\langle \texttt{let}\ x = n\ \texttt{in}\ c, \sigma\rangle \to \langle x := n\ ;\ (c\ ;\ x := \sigma(x)), \sigma'\rangle$$

---

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

- **0 pts** Correct