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.

Solution One of Hoare's claims that I strongly agree with is "a good programming language will encourage and assist the programmer to write clear self-documenting code." For many of my projects, a large amount of initial effort is spent trying to understand code someone else wrote that I would like to build upon. This is especially difficult with code for research projects, since this code is sometimes not kept to the standards of style and review enforced in industry. After working with a few codebases, I've found that the coding language used can affect the readability of the code, making up for the author's potential lack of attention to this aspect.

The baseline readability of different coding languages can be seen when comparing C++ and Python. The explicit and verbose variable type, function signature, and class field declarations found in C++ but not Python can make it easier to reason about someone else's C++ code rather than their Python code. For example, when I had to build upon some distributed systems research written in C++, it was straightforward to see what information they were tracking for each request by looking at their custom structure definitions in their header files. On the other hand, for many machine learning projects, I've had to build on someone else's Python code, and the lack of type declarations can make it difficult to tell when a variable is a tensor, list of lists, or NumPy array. It's also sometimes hard to tell what functions are returning without return types, especially since functions can have multiple returns with different types of data in Python. Python may be faster to code in, but C++ offers hints that make it easier to read.

Continuing with the topic of program readability, I somewhat disagree with Hoare's claim that "the design of a super comment convention is obviously our most important concern." Comments are indeed useful, but not as useful as well-chosen variable, function, and class names, as well as small functions with limited, clear goals. The issue with comments is that to stay consistent with the code, they need to be actively maintained by the programmer. Since code is constantly being redesigned and refactored during development, this can lead to lazy, omitted, or misleading comments. Comments should supplement programs with good naming and structure, not patch up bad ones.

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.

Solution To introduce an integer division operator to IMP, we must define a new rule of inference as follows:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad \langle e_2 = 0, \sigma\rangle \Downarrow \mathit{false}}{\langle e_1/e_2, \sigma\rangle \Downarrow (q \text{ s.t. } n_1 = qn_2 + r \text{ where } q, r \in \mathbb{Z}, 0 \le r < n_2)}$$

This rule takes into account two factors of integer division. First, the hypothesis ensures that division by an expression that evaluates to 0 is never valid. Second, the construction of q gives a unique integer result, since r is constrained to be $0 \le r < n_2$.

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.

Solution The rule of inference for `let` should be:

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

The rule executes c in σ with x assigned to e, records the resulting state, and reassigns x to its initial value.

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.

Solution The set of redexes should be extended to $r ::= \ldots \mid \texttt{let}\ x = n\ \texttt{in}\ c$, where n is a number, not an expression. The reduction rule for this redex should be

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

The set of contexts then needs to be extended to $H ::= \ldots \mid \texttt{let}\ x = H\ \texttt{in}\ c$ to allow for expressions to be used for x.

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

- 0 pts Correct

Peer Review ID: 63245417 — enter this when you fill out your peer evaluation via gradescope