All subsequent answers should appear after the first page of your submission and may be shared publicly during peer review.

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.

Something in Hoare's paper that I experience frequently in my programming experience is the need for good documentation of my code. Honestly, more important than writing a correct program is writing a program that can be easily read and understood by someone else. To assert it explicitly, bad code that works is worse than good code that doesn't. The reason here is that when (not if) something goes wrong with the bad code, it is significantly harder to fix. On the other hand, good code that doesn't work now is much easier to debug and get working (especially as a team).

One of Hoare's points that I truly agree with is the overarching idea that these PL concepts don't much matter when a program works as intended. Rather, they are only important when something goes wrong. The reason for this is clear: if the language is simple in design and can be recompiled quickly, errors are much easier to locate and remediate. If the code works correctly, the programmer doesn't need to give any thought to whether compilation took a long time or the code was easy to write and read.

Lastly, one point of Hoare's that I disagree with is the need for and importance of typing in PL. Python is one of the most useful and widely used programming languages, and it forgoes the constraints of variable typing almost entirely. In addition, this leads to some useful properties of the language, which outweigh the burden of encountering type errors. First and foremost is the speed with which Python can be written, given that paying tedious attention to types is rendered moot. A base assumption of this is the programmer is skilled enough to be able to keep track of what is happening in his program, but Hoare has already argued that programming languages should have properties that are catered to those who have mastered them.

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.

For this, we need to create a rule of inference for division. Naively, we posit the rule to be

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

However this does come with 2 complications that we must account for. The first is that this is integer division, and the second is that division by 0 must be expressly disallowed. Accounting for divide by 0 is as simple as adding the hypothesis

$$\neg n_2 = 0$$

Next, we handle the fact that we are expressly doing integer division, meaning our result and all intermediate results must be integers. Here, we use the fact that integer division is a well defined mathematical concept (recursive subtraction is one such way to define it). As such, we simply change the operation being done on n_1 and n_2 (we'll represent it as $/_{int}$). Finally, we come up with the rule for division

$$\frac{< e_1, \sigma > \Downarrow n_1 \qquad < e_2, \sigma > \Downarrow n_2 \qquad \neg n_2 = 0}{< e_1 / e_2, \sigma > \Downarrow n_1 \ /_{int} \ n_2}$$

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.

We want to create a rule that results in the following conclusion

$$< \text{let } x = e \text{ in } c, \sigma > \Downarrow \sigma'$$

We take note of the fact that if the variable that declared as a part of the Aexp in the let statement must retain its original value from σ at the end of execution. All other variables that are changed will have those changes persist, and as such we don't need to account for them. As such, we simply need to assert that $\sigma(x) = \sigma'(x)$

Our resulting rule is

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

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 seek to create redexes, contexts, and reduction rules to account for the command `let` $x = e$ `in` c. We start with the redexes. We already have a redex for evaluating $x = e$ to n, so once that is reduced we can simply evaluate c. Thus, we get the redex:

$$\texttt{let } \sigma[x := n] \texttt{ in } c$$

From here, we have to account for this new redex in our reduction rules. The next step after this is to execute the command, so we get the reduction rule:

$$< \texttt{let } \sigma[x := n] \texttt{ in } c, \sigma > \rightarrow < c, \sigma >$$

Now, all that remains is adding relevant contexts to get us from a full let statement to the redex which we have previously defined. First, we will need to evaluate the expression e. Second, we will need to update the variable. These contexts can be shown as:

1. `let` $x =$ • `in` c
2. `let` • `in` c

This is all that is necessary to account for the let command in our small step semantics

Exercise 1C. Language Feature Design, Coding. Download the Homework 1 code pack from the course web page. Modify `hw1.ml` so that it implements a complete interpreter for IMP (including `let` and `print`). Base your interpreter on IMP's large-step operational semantics. The `Makefile` includes a "`make test`" target that you should use (at least) to test your work.

Modify the file `example-imp-command` so that it contains a "tricky" terminating IMP command that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

Submission. Turn in the formal component of the assignment (1F-1 through 1F-5) as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment (1C) via the `autograder.io` website.

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

- **0 pts** Correct