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.

Hoare argues that "a good programming language should give assistance in expressing what it is intended to accomplish" and that "a good programming language should reduce as far as possible the scope for coding error; or at least to guarantee that such errors can be detected by a compiler or cheaply detectable at run time." In my understanding, this means that a language should be able to express "pre/post-conditions" for a command. Ideally, the pre/post-conditions should be checked at compile-time or very easily checked during run-time.

I think in practice, language features that support this ideal fall into two categories. The first one is "types", which are compile-time checkable conditions. The second one is "assertions", which are run-time checkable conditions. "Assertions" do not require any smart design in the language itself, so I will focus on "types". Many PL research try to come-up with a type system that enforces programmers to write a "safe/correct" program whose safety/correctness is guaranteed by a compiler. This ideal is really nice because programmers are usually lazy and they need some discipline (a compiler) to force them write good code. However, languages with too strict typing may be difficult to use, which may also violate Hoare's argue for "simplicity". I think a good programming language should make an appropriate trade-off between simplicity and the the power of type system.

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: We need to update the syntax of Aexp to include $e_1 \div e_2$ for $e_1, e_2 \in$ Aexp, and add the rule: $\dfrac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e1 \div e2, \sigma\rangle \Downarrow q}$ where $q \in \mathbb{Z}$ is the quotient of $n_1 \div n_2$.

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.

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

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:

$$\frac{\langle e, \sigma\rangle \to \langle e', \sigma\rangle}{\langle \text{let } x = e \text{ in } c, \sigma\rangle \to \langle \text{let } x = e' \text{ in } c, \sigma\rangle}$$

$$\frac{\langle x, \sigma\rangle \to \langle v, \sigma\rangle \quad \langle c, \sigma[x := n]\rangle \to \langle c', \sigma'[x := n]\rangle}{\langle \text{let } x = n \text{ in } c, \sigma\rangle \to \langle \text{let } x = n \text{ in } c', \sigma'[x := v]\rangle}$$

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.