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.

1F-2 Answers I found Hoare's essay fascinatingly topical and relevant, despite the fact that it's about programming languages written on punch cards and comments on "newer languages" with "/* */"-style comments (i.e. B, which is the same age as my parents). I disagree with his arguments about references, however. In my experience, references have been useful–as in our other reading, the one on JavaSchools, they "separate the men from the boys". Pointers are fast, and in classes like EECS 281 and in the real world speed is a necessity. Passing arguments to programs by reference is also useful because it avoids duplicating large data structures. This may not be wholly what Hoare was referring to when he references references; in my experience, I've worked with mainly C++ and Python and with relatively-small programs, where references are separated enough from the rest of the machine's memory that I don't need to worry about trashing the BIOS of the computer with a missed pointer. Hoare mentions memory references messing with instruction pipelines–but in a modern computer like the one I am typing on, I am typically working so many levels of abstraction up from the hardware that missing even hundreds of thousands of cycles is barely noticeable, and the speed which not copying structures can bring is much more noticeable.

However, I wholeheartedly agreed with his comments on simplicity. In my experience coding, only understanding part of a language has always been confusing–whether it's the stripped-down version of Java with no UI instructions which AP Computer Science taught me or Javascript in general, I've always been left scrambling. Telling me to type "public static void main(String[] args)" with no reason why for even the simplest Hello World just leads to confusion when one of those keywords is left out and the compiler complains. It's not necessary to have everything in the Python documentation memorized, but I understand the core of the language and it's currently my go-to. However, if taught poorly, even simple design can be insufficient for comprehension–I once had to tutor an older cousin through Python when his introductory class taught try/except before loops, and I'm not sure he could evaluate booleans even at the end.

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.

1F-3 Answers This would be integer division, as Aexp always takes and evaluates to integers. There would be two numbers, a divisor and a dividend, and in most common forms of integer division the result would be the quotient. We can consider integer division as follows: $a/m = q$ where $a = mq + r$ and r, the remainder, is the smallest possible positive remainder, between 0 and $m - 1$ (This works on negative numbers, too; -10 / 4 = -3, with remainder 2). To get q, the quotient, we take $\lfloor a/m \rfloor$. In the case of division by zero, we would need to code in some kind of error; r <m but in the case $a = 0q + r$ r would need to be greater than m = 0. We would add two operational semantics rules of inference.

$$\frac{\langle e1, \sigma\rangle \Downarrow n1 \qquad \langle e2, \sigma\rangle \Downarrow 0}{\langle e1/e2, \sigma\rangle \Downarrow Error}$$

.

$$\frac{\langle e1, \sigma\rangle \Downarrow n1 \qquad \langle e2, \sigma\rangle \Downarrow n2}{\langle e1/e2, \sigma\rangle \Downarrow \lfloor n1/n2 \rfloor}$$

.

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.

1F-4 Answers To deal with let, we need to evaluate this in the same manner that we deal with the other commands in class. Specifically, changes to the variable x need to be undone when the scope of let ends, but changes to other variables need to be maintained. We do this by re-modifying the σ' after running the command c.

$$\frac{\langle \text{ let x = a in c}, \sigma(x == d)\rangle \Downarrow \sigma'[x := a] \qquad \langle c1, \sigma\rangle \Downarrow \sigma'}{\langle\langle c1, \sigma[x == a]\rangle \Downarrow \sigma'[x == a]\rangle \Downarrow \sigma''[x := d]}$$

Peer Review ID: 63248759 — enter this when you fill out your peer evaluation via gradescope

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.

1F-5 Answers To account for let, it will need contexts; specifically, the ability to persist potential variable assignments across context. The reduction rule for let x = a in c can be written as follows: $\langle \text{let x = a in c}, \sigma\rangle \rightarrow \langle placeholder := x; x := a; c; x := placeholder, \sigma\rangle$. Like while, we temporarily extend the program to allow us to simplify it further later.
The redex is extended to include the left side which is simplifiable according to this rule: r ::= ...|let x = a in c. We add a context that the let must be executed before the c it contains, so that the variable x is correct in context: H ::= ...|H in c. From the redex, we determine that we don't need to add further contexts to H and that we have the extant placeholders (for example, x := H and H; c are already placeholders specifying that we simplify assignments before assigning and that we work commands top to bottom).

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