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.

I think overall Hoare's analysis is helpful and meaningful. Despite it being written almost 50 years ago, many of the points discussed still hold today. Of course, over the years, some of these points will naturally be less relevant. One interesting area is his discussion of documentation. I think that documentation is just as important as it was back then; however, the various aspects of documentation have changed. The necessity of documentation is just as relevant today, while the need for a robust comment design is less important when it comes to the design of the language.

I think the points made regarding documentation are spot on even today. Documentation should be encouraged throughout the development process, not as an after thought to satisfy code reviews or help new team members get acclimated. Additionally, the language should help make it easier to document the code. This happens less frequently than we'd like. One good example is how Java generated documentation using the Javadocs schema. This encourages programmers to follow documentation guidelines to minimize their effort and produce beautiful documentation that one can browse in the web. This is one of the best feature of Java in my opinion and shows how the language can help encourage developers to document their code.

I think his points made about comment styles, however, is not quite as important as some of the other points. A lot of this has to do with the evolution of technology since his report was written. Most modern languages support single and multi-line comments using a short series of special characters which makes typing them in not a huge hassle. Additionally, he mentioned how multi-line comments can lead to typos which comment out extremely large blocks of a program (when a trailing */ is omitted in a C++ program for example). While this is always unfortunate, these days, thanks to the faster speed of compilers and help from amazing text editors and IDEs, these problems can be quickly addressed. IDEs these days often have syntax highlighting and comment visibly stand out from the rest of the code which makes spotting large commented blocks much easier. Thus, I argue that it is not quite as important anymore to have a superb comment style in a programming language compared to some of the other features such as documentation and debugging tools provided.

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.

To do this, we need to introduce the following rule of inference:

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

Here the division operator refers to integer division which means we simply divide the two numbers and truncate any decimals to keep the answer as an integer. Additionally, the division operation will return undefined if e_2, and by that logic n_2, is 0.

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.

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

We evaluate the expression e and set x to this new value of in the new state(σ'). Then we evaluate the command c in the new state before getting a second new state(σ''). Finally, we set x back to the original value once we're done.

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.

Redex:

$$\text{let } x := n \text{ in } c$$

Reduction Rule:

$$\langle \text{let } x := n \text{ in } c, \sigma \rangle \rightarrow \langle x := n; c; x = \sigma(x), \sigma \rangle$$

Context:

$$H ::= \bullet | x := H; c; x := H[\sigma[x]]$$

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).

Submitted to Autograder :)

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