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's article mentions a few high level properties or features that all modern programming languages should follow or support. While I can see aspects of all of them in the languages I use in my own work, the one that was particularly interesting to me was fast translation. Learning to program in C++ has shown me the importance of fast compilation right from the beginning. Any small change requires a new compilation to run the program. C/C++ has been around for a long time now and now has highly optimized compilers to build the executables. I have seen this personally when writing low level multi threaded code for my operating systems class. These were complex programs that required many hours of debugging and changing code. Every attempt to make a change or even run using gdb required a compilation of some sort. If this process was not optimized I don't believe we would have been able to rely so heavily on this process of making small changes and seeing changes reflected in the output. Another reason I believe that fast translation is crucial is because of test driven development. I have been taught and urged to work in a test driven environment. This means that I write small test cases and then write code to pass those test cases, constantly adding code and compiling to check that the program functions correctly at every step. This process depends heavily on compilers working efficiently so that programmers spend minimal time waiting and most of their time on actual development. Fast translation and efficient compilers are definitely an important part of programming languages, and has been something that I have relied on majority of my computer science career.

While most of the code and languages I've worked with compile fast, when I interned last summer I worked on the Android team where I spent my time working in Android Studio and Kotlin developing for a large application. This was my first exposure to mobile development and it was a shock to see that Gradle required around 10-15 minutes to compile and build. This was because to see changes reflected on an emulator or physical phone, the entire application needed to be compiled together again as a package. We used the latest software, newest technologies, and followed all documentation and guidelines to write efficient code. The compilation would still take a significant amount of time. This makes me realize that while fast translation and compilation is definitely important and ideal, there may be times when we must trade it for accuracy, reliability, and robustness of the program and builds. The entire Android community deals with relatively slow build times and while everyone wishes that we could speed this up, there is no alternative that can provide this along with all the features we already have and enjoy in the development process. Fast compilation is definitely something we all would love to have, but in some cases we must prioritize other properties to ensure the best development process.

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.

There will be 2 rules to cover 2 cases: the general case and division by 0. I will define division by 0 to equal 0 since it is otherwise undefined.

$$\frac{< e_2 = 0,\ \sigma > \ \downarrow \ False \qquad < e_1,\ \sigma > \ \downarrow \ n_1 \qquad < e_2,\ \sigma > \ \downarrow \ n_2}{< e_1/e_2,\ \sigma > \ \downarrow \ \lfloor n_1 / n_2 \rfloor}$$

$$\frac{< e_2 = 0,\ \sigma > \ \downarrow \ True}{< e_1/e_2,\ \sigma > \ \downarrow \ 0}$$

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 <c, σ> ⇓ σ' 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.

$$\frac{< x := e;\ c,\ \sigma > \ \downarrow \ \lambda \qquad < x := \sigma(x),\ \lambda > \ \downarrow \ \gamma}{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$< Let\ x\ =\ e\ in\ c,\ \sigma >\ \downarrow\ \gamma$

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.

$H ::= ... \mid let\ X\ =\ H\ in\ C$
$R ::=\ ... \mid let\ X\ =\ e\ in\ C$
$< Let\ x\ =\ e\ in\ C,\ \sigma >\ \rightarrow\ < x ::=\ e; c; x = \sigma(x),\ \sigma >$

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

- **0 pts** Correct