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.

Hints on Programming Language Design is a comprehensive review of Hoare's thoughts on programming design. In the introduction of the paper, Hoare explains "I fear each reader will find some of my points wildly controversial; I expect he wil find other points that are obvious and even boring; I hope he will find a few points that are new and worth pursing." I found this prediction to be correct as I read through this paper, as many of the points rang true to my experiences with programming languages. More specifically, Hoare's point on prioritizing modularity over simplicity was especially pertinent. In the paper, Hoare describes how many programming languages have focused on modularity over simplicity of design. With this focus on modularity, many programmers only understand a portion of the language and may accidentally access a part of the language they are not aware of by accident, leading to to unexpected bugs. I have experienced this problem with modularity over simplicity with Python. For the past two semesters, I have served as a graduate student instructor for a class that utilizes Python. Since Python's different libraries and functionality are so vast, that I can only understand the portions of it relevant to the project. This becomes problematic when trying to help students solve problems, who may have figured out a different implementation strategy. While this strategy is still valid in Python, it may be from a part I'm not aware of so I can not offer meaningful help to the students. With a sprawling language like Python, modularity is often phrased as one of its benefits, but I have found I agree with Hoare that too much of an emphasis on modularity can lead to more problems then a smaller, simpler language.

Although I agreed with the bulk of Hoare's paper, one of his recommendations I disagreed with some of his assertion. One key assertion that I disagreed with is his belief that there should be no form of automatic type transfer, except when defined by the programmer. I understand his logic, that automatically converting types leads to possible confusion from the programmer when this unexpected type transfer results in a bug. There is an exception to this rule with respect to numerical representations. Often, when working with calculations, both integers and floating point data types are used in the same calculations. Under Hoare's recommendation, trying to add a floating point number and an integer would result in an error, unless the programmer had explicitly cast one of the operators to be the other type. While this could add some clarity, adding casts at every point in a complex calculation would result in a large amount of code bloat. Python automatically casts integers to floating point data when they are in an operation with floating point numbers, and this eliminates the need for casts without loss of accuracy in casting a floating point digit to an integer. In my experiences in helping students with computation heavy problems in the class I teach,

removing the possibility of loss of accuracy in numeric calculations helps cut down on the space of possible bugs. Overall, this practice of Python automatically casting a floating point digit to an integer increases the simplicity and usability of Python, compared to if Python did no automatic casts.

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.

Extend Aexp as follows:
$e ::= n$ for $n \in \mathbb{Z}$
...
$\mid e_1/e_2$ for $e_1, e_2 \in$ Aexp

The operational semantics must be expanded to include a rule to define what division does. This step will describe integer division, which cuts off the decimal point values of any divided value with a nonzero decimal value. This operation can be done by taking the floor value of the numeric result of dividing the value of e_1 and e_2. Division is only defined for any expression e_1 and any expression e_2 where e_2 does not evaluate to 0. Division is left undefined for $\langle e_2, \sigma \rangle \Downarrow 0$.

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \ \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1/e_2, \sigma \rangle \Downarrow floor(n_1/n_2)}$$

For $n_2 \neq 0$. Integer division is not defined for $n_2 = 0$.

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.

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

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.

Redexes
$r ::= x$
$\mid n_1 + n_2$
...
$\mid$ `let` $x = n$ `in` c

Contexts
$H ::= \bullet$
$\mid n + H$
...
$\mid$ `let` $x = H$ `in` c

Local reduction rule for `let`
$\langle$`let` $x = n$ `in` $c, \sigma\rangle \to \langle c; x := \sigma(x), \sigma[x := n]\rangle$

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

- **0 pts** Correct