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.

In favor

In some of the hints that he gives, he points to the way comments are handled and the how the omission of END notation leads to sections of codes disappearing and overflowing to another section of code. I agree with this idea as I avert using /* */ for comments in C++ and a bit cautious using python as it does not require explicit declaration of different sections of code (i.e. it uses spaces to differentiate sections of code). I find both practices very prone to error as I would comment out huge sections of code that I did not mean to, and group code together without an easy way of check if the code is spaced the right amount.

Against

I disagree with his claim that pointers should be removed due to their complexity. The 2 languages that I'm most comfortable with are C++ and Python. And every time that I transition from C++ to Python, I'm always stumped by how I my code would work under the hood when I change the assignment of a variable. However, it's much more clearer with C++ as I would just change the address of the pointer. I understand the argument that this practice leads to more errors but giving the option to the programmer aids in writing efficient code.

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.

Division is unique in that it cannot be divided by zero, while other operations does not impose a limit in the operands. Therefore, for a division operator I would check if the denominator is a zero before the operation is performed.

$$\langle e_1/e_2,\sigma\rangle \Downarrow n \rightarrow \frac{\dfrac{\langle e_2,\sigma\rangle \Downarrow n_1}{\langle e_2 != 0,\sigma\rangle \Downarrow \text{True}} \quad \langle e_1,\sigma\rangle \Downarrow n_2}{\langle e_1/e_2,\sigma\rangle \Downarrow n_1/n_2}$$

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 1 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.

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

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.

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

Context & redex:

$H ::= \langle x := \bullet;\ c;\ x := \sigma(x), \sigma \rangle$

$r ::= e$

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!

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

- **0 pts** Correct