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.

Answer: Hoare emphasizes that simplicity in language design leads to clarity, maintainability, and robustness. A simple language minimizes cognitive load for developers, allowing them to focus on solving problems rather than wrestling with the complexities of the language itself. This is something I've seen play out in my own experience with languages like Python. Python's simplicity allows developers to write clean, easy-to-read code. It abstracts away many of the low-level details, letting programmers focus on logic and problem-solving instead of worrying about memory management or language syntax intricacies. This simplicity has made Python one of the most popular languages in data science, web development, and rapid prototyping.
Hoare also emphasizes that control structures in programming languages should be simple and few. While simplicity is valuable, the idea of minimizing control structures can be limiting in certain contexts. For instance, in modern languages like Scala or Haskell, advanced control structures like pattern matching or list comprehensions lead to more expressive, concise, and readable code. Scala's Pattern Matching is an example of a control structure that allows developers to easily deconstruct data structures in a way that's more elegant and intuitive than a series of if-else statements or even switch cases. While Hoare's principle may encourage more basic structures, this advanced approach significantly enhances clarity in some scenarios.

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.

Answer: We need to add an error value for zero divison.
$$\frac{\langle e_1,\sigma\rangle \Downarrow n_1 \quad \langle e_2,\sigma\rangle \Downarrow n_2 \quad \neg(n_2{=}0)}{\langle e_1/e_2,\sigma\rangle \Downarrow n_1/n_2} \qquad \frac{\langle e_1,\sigma\rangle \Downarrow n_1 \quad \langle e_2,\sigma\rangle \Downarrow n_2 \quad n_2{=}0}{\langle e_1/e_2,\sigma\rangle \Downarrow error}$$

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):

Questions assigned to the following page: 5 and 4

```
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.

Answer: $\frac{\langle e,\sigma\rangle \Downarrow n_1 \quad \langle c,\sigma[x:=n_1]\rangle \Downarrow \sigma'}{\langle let\, x = e\, in\, c,\sigma\rangle \Downarrow \sigma'}$

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.

Answer: Redexes: let x = r in c
Contexts: let x = • in c
Reduction rules: $\langle let\, x\ =\ r\, in\, c, \sigma\rangle \rightarrow \langle c, \sigma[x = r]\rangle$

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.

Peer Review ID: 301740613 — enter this when you fill out your peer evaluation via gradescope