All subsequent answers should appear after the first page of your submission and may be shared publicly during peer review.

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 1F-2: I agree with Hoare that the readability of a programming language is more important than its writeability. Throughout my internships, I've found that projects change hands frequently, not to mention the high turnover rate that plagues many tech teams. For this reason, there is often significant time spent reading code that you did not write so that you can debug it and make improvements. High readability would add greatly to a programming language's utility, because it would help speed up code production by reducing the amount of time needed to read and understand an inherited codebase.

However, I don't think that there needs to be such a great emphasis on comment conventions. Certainly, there are more and less efficient ways to define a comment in a programming language, but the emphasis, I think, should be on more critical matters. Personally, when I have accidentally misplaced a comment closing symbol, it has been a simple problem to fix. Fixing problems within the code itself has always far outweighed any problems with comments, both in time spent and amount of developer frustration, which tells me that is where the emphasis should lie.

To summarize, Hoare covers a variety of important points, all of which are objectively good ideas about programming language design. However, I think that there are some points that might not need to be given as much weight as the others, given that the list seems to be rather exhaustive. In other words, selectively choosing a few of the most critical ideas might result in an overall better programming language design, as trying to incorporate all of Hoare's points might generate unnecessary complexity.

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 1F-3 For integer division, the answer is usually rounded down, and I will follow that convention in this problem (for example, $5 \div 3$ would be rounded down to 1. To write this as a rule, we create a while loop, where given the division $a \div b$, we subtract b from a until a is less than b, keeping track of how many times we subtract b in some additional counter variable that we can call c. In the following, we will assume that c is initialized to

0. This can be expressed as the following rules of inference:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 \div e_2, \sigma\rangle \Downarrow n_1 \div n_2}$$

$$\frac{\langle \text{while } b \leq a \text{ do } c := c + 1; a := a - b, \sigma\rangle \Downarrow \sigma'}{\langle n_1 \div n_2, \sigma\rangle \Downarrow \sigma'(c)}$$

The special, error case that needs to be considered with integer division is when the divisor (or denominator, when written in fraction form) is zero. To indicate the the division cannot be performed, an error can be returned. This can be expressed as the following rule of inference:

$$\frac{\langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1 \div e_2, \sigma\rangle \Downarrow \text{error}}$$

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):

```
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 1F-4

The new rule consists of four main parts. First, a temporary variable *temp* is created to hold the value of x prior to any part of the `let` command being executed. Second, the $x = e$ assignment is executed. Third, the command c is executed. Finally, x is returned to its original value using the temporary variable *temp* created in the first step. Because the rule

is rather long, I wrote it in parts.

$$A : \langle t := x, \sigma\rangle \Downarrow \sigma[t := \sigma(x)]\rangle$$
$$B : \langle x := e, \sigma[t := \sigma(x)]\rangle \Downarrow \sigma[t := \sigma(x),\ x := n]\rangle$$
$$C : \langle c, \sigma[t := \sigma(x), x := n]\rangle \Downarrow \sigma'\rangle$$
$$D : \langle x := t, \sigma'\rangle \Downarrow \sigma'[x := \sigma(x)]\rangle$$

$$\frac{A\ B\ C\ D}{\langle \text{let } x = e \text{ in } c, \sigma\rangle \Downarrow \sigma'[x := \sigma(x)]}$$

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.

Answer 1F-5

$H ::= ...| \ t := H;\ x := e\ ;\ c\ ;\ x := t$
$| \ x := H;\ c\ ;\ x := t$
$| \ c\ ;\ x := t$
$| \ x := H$

$r ::= ...| \ x \mid x := n \mid skip;\ x := e \mid skip;\ c \mid$

$\langle \text{let } x := e \text{ in } c, \sigma\rangle \rightarrow \langle t := x; x := e; c; x := t, \sigma\rangle$
$\langle e, \sigma\rangle \rightarrow \langle n, \sigma\rangle$ where e evalutes to n
$\langle x := n, \sigma\rangle \rightarrow \langle \text{skip}, \sigma[x := n]\rangle$
$\langle c, \sigma\rangle \rightarrow \langle \text{skip}, \sigma'\rangle$

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.

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

- **0 pts** Correct