2. 1F-2

Solution: An aspect of Hoare's *Hints On Programming Language Design* that I can relate to as a programmer is the objective of modularity, or the fact that a programmer who doesn't fully know the ins and outs of a language can still get by with only a limited understanding of the language. A benefit to this is that it is very easy to pick up new languages as an experienced programmer, which I've been able to do (such as picking up Python with C++ knowledge) because a lot of the keywords and syntax overlap. However, the new language may have a more efficient way to achieve something that I may not know about. I think that this idea of modularity makes it challenging for new language designers to innovate too far from the norm of already developed languages. Having clear documentation can certainly be a step forward in encouraging programmers to gain a full, deeper understanding of a new language.

One of Hoare's points that I agree with is that language designers should focus on designing a readable language, as well as one that encourages programmers to write good comments. Since human programmers are frequently collaborating on writing code, it's important for them to efficiently read other people's code, and this includes documentation. This improves maintainability and reduces errors for larger-scale projects. Although the computer is indeed the last to execute the code, humans are ultimately the ones behind creating software, and the design of the language should cater to their cognition.

One of Hoare's points that I would argue against is that automatic type conversion is dangerous. Although I somewhat agree that it may be beneficial on the compiler level in terms of quickly catching type-matching errors, there is evidence of many programming languages that do not require explicit types such as Python and Javascript. These languages still perform well and safely, and enhance the programmer's efficiency, as well as the flexibility and conciseness of code. While there may be some merit to Hoare's argument such as the fact that dynamic typing encourages sloppy programs, I think that it may actually have better benefits if the language is designed carefully.

3. 1F-3

Solution: First, we add a new division operator to the Aexp abstract syntax:

$$e1/e2 \text{ for } e1, e2 \in Aexp$$

Then, we introduce the following big-step rule:

$$\frac{\langle e_1, \sigma\rangle \downarrow n_1 \quad \langle e_2, \sigma\rangle \downarrow n_2 \quad n_2 ! = 0}{\langle e_1/e_2, \sigma\rangle \downarrow n_1/n_2}$$

Questions assigned to the following page: 3, 4, and 5

In the case that the denominator is zero, the conclusion results in an error, so we introduce the rule:

$$\frac{\langle e_1, \sigma\rangle \downarrow n_1 \quad \langle e_2, \sigma\rangle \downarrow 0}{\langle e_1/e_2, \sigma\rangle \downarrow error}$$

4. 1F-4

Solution: The following rule extends the IMP language to include the `let` command:

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

This evaluates e to n in state σ, then evaluates c in the original state σ with x replaced with the new value n, resulting in a new state σ'. The final state returned is σ' with x replaced to its original value from σ.

5. 1F-5

Solution: To extend the set of redexes, contexts and reduction rules for the `let` command, let $x = e$ in c, we first add a new redex:

$$r ::= \text{let } x = e \text{ in } c$$

Next, we add a new context:

$$H ::= \text{let } x = H \text{ in } c$$

Finally, we add two reduction rules. One to evaluate the expression e:

$$\langle H[r], \sigma\rangle \rightarrow \langle H[n], \sigma\rangle$$

The next to execute the command with the updated value for x:

$$\langle \text{let } x = n \text{ in } c, \sigma\rangle \rightarrow \langle c, \sigma[x := n]\rangle$$