Exercise 1F-2. After reading Hoare's Hints on Programming Language Design, I am astonished at how well it holds up. Almost all of the recommendations should still be heeded by language designers today. Much of his advice was subsequently followed, to the benefit of modern languages (e.g. comment conventions, context free grammars, operator overloading, avoidance of pointers, case statements, type systems, variable scoping, and more). Some of his advice has been ignored by some languages, but it is generally to their detriment (e.g. lack of type system in Javascript, C++ not being context free, Python ignoring performance because computers are fast). Seeing that so many PL design principles were already understood in 1973 makes me question why a language like javascript even exists.

The piece of advice I like the most is that features should not be added just because they are easy to implement or a clever use of the orthogonality of other features. He warns that these features will be abused by clever programmers at the cost of understandability. I have seen this frequently with C++ with features such as overloading the function call operator. C++ is full of examples of things which seem elegant at the surface, but which rely under the hood on clever language features that make them incomprehensible as soon as you try to predict how they will work in edge cases.

My main criticism is that I think Hoare overemphasizes simplicity. Simplicity is a noble goal, but he prioritizes simplicity above everything, and I think this is a mistake. The essence of building a useful language is accepting good trade-offs; if the benefit of complexity is large enough, giving up simplicity may be justified. An example of this is Rust's memory management system. It adds complexity that makes Rust much harder to learn and much harder to prototype in, but it is nonetheless a good trade-off. It was only implemented after decades of experience with the dangers of manual memory management in C and C++. With experience and hindsight, programmers have decided that the complexity of a borrow checker is far outweighed by the usefulness of it's memory safety guarantees. In general, once a domain is well enough understood, it is possible to give up simplicity in the areas where complexity is most helpful. Type systems are another example of a good tradeoff. Hoare argues for the the use of type systems, but he does not generalize this to other good tradeoffs. This is understandable, because Hoare could not rely on the decades of software development hindsight that choosing good tradeoffs requires, but I think it is a place the paper does not hold up as well.

Exercise 1F-3. To extend IMP with a division operator, we just need to add a division operator "/" to the syntax, and add an appropriate inference rule to define it's semantics. Since division by zero is not defined for integer division, we will decide to define it as zero.

"/" means integer division.

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1/e_2, \sigma\rangle \Downarrow \text{if } n_2 = 0 \text{ then } 0 \text{ else } n_1/n_2}$$

Exercise 1F-4.

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

Exercise 1F-5.

Redexes:
r ::=
...old redexes...
| let $x = n$ in c

Contexts:
H ::=
...old contexts...
| let $x =$ H in c

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