Exercise 1F-2. Language Design

While reading Hoare's paper, one theme that struck me that some of his specific criticisms lack relevancy, but his general principles still are important for language design.

His principles are still poignant. For example, simplicity remains important today. The success of Python is at least partially due to its simple syntax. U.S. government agencies have encouraged the use of "memory-safe" languages, which seems to mean to conduct new development using simpler languages that abstract memory allocation away from the programmer. Simplicity also continues to motivate trends in functional programming found in languages like Haskell or Elixir. His discussion of readability continues to be an actively-discussed topic in software engineering, and a programming language should help support this. While the meaning of the word seems to have changed from his paper, his focus on 'security', is still important in the sense that programming languages should help users detect errors.

Some of Hoare's advice has diminished in relevance as programming languages have advanced. His specific concerns about security seem to hinge around lacking debugging tools in production, which could lead to reliability issues. The issue of runtime reliability between production and development code seems to have been addressed. He also suggests that the debugging modules may take up too much storage space to fit on a machine. This is another issue that I believe has been solved for the vast majority of programs. He also cautions against an optimizing compiler as being too slow. However, compilers seem to have improved to the point where applying a set-series of optimization passes (for example with the -O3 flag in a C++ program) require negligible additional compile time and do not introduce concerns with program reliability. More generally, his criticism is focused on deficiencies in Algol 60 and FORTRAN, which have less wide-spread use these days. For example, he dedicates significant about a page to talking about ideal comment conventions. Most programming languages in use have found ways to incorporate both single-line comments and multi-line comments in ways that already incorporate his points of advice.

Exercise 1F-3. Simple Operational Semantics

First, the abstract syntax will need to be updated for Aexp to include syntax for division. Focusing on operational syntax though, the main changes are to define two new rules of inference: one for handling division (not by zero), and a second for handling division by zero. For handling normal (not by zero) division:

$$\frac{\langle A_1, \sigma \rangle \Downarrow v_1 \quad \langle A_2, \sigma \rangle \Downarrow v_2 \quad v_2 \neq 0}{\langle A_1 / A_2, \sigma \rangle \Downarrow v_1 \div v_2}$$

For handling division by 0:

$$\frac{\langle A_1, \sigma \rangle \Downarrow v_1 \quad \langle A_2, \sigma \rangle \Downarrow 0}{\langle A_1 / A_2, \sigma \rangle \text{ undefined}}$$

Exercise 1F-4. Language Feature Design, Large Step

To capture the idea that the change from e is only applied to c in a local scope, we can define the following rule of inference. This rule concludes with the state σ' but only after removing the local-variable x removing from itself:

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

Exercise 1F-5. Language Feature Design, Small Step

The grammar for redexes can be extended to capture the reduction for the *let* statement:

$$\begin{aligned} r ::=& \text{ Existing grammar rules} \\ &\mid \text{let } x = e \text{ in } c \end{aligned}$$

A new local reduction rule can be added:

$$\langle \text{let } x = e \text{ in } c, \sigma \rangle \to \langle c, \sigma[x \mapsto v] \rangle \quad \text{if } e \to v$$

However, this rule does not capture that the state σ for x needs to return to it's prior condition from before the execution of c in order to capture the local scope. I am not sure how to represent this with small-step semantics.

The grammar for contexts can be extended to account for *let* through a rule that shows that the expression e in the *let* statement can be evaluated in its own context:

$$\begin{aligned} H ::=& \bullet \\ &\mid \text{Existing grammar rules} \\ &\mid \text{let x = H in c} \end{aligned}$$