Exercise 1F-2

Though it was written nearly 50 years ago, much of Hoare's *Hints on Programming Language Design* still feels strikingly relevant. The three pillars of programming are still design, documentation, and debugging, and I feel like some aspects of programming would be easier if programming languages did a better job of supporting these pillars (especially documentation and debugging).

One of Hoare's opinions that I support is his general dislike of programs that provide large data structures with built-in operations. He reasons that the optimal implementation of a data structure may be different for different use cases, but that including a built-in data structure that is significantly easier to program with (as a language feature) will shoe-horn programmers into using it even if it isn't optimal. He says that one solution to this problem would be "extensible" languages, which were generally unsuccessful at the time. Nowadays, however, I see C++ as a great example of such an extensible language. The only language-defined types in C++ are the primitives – the rest of the standard library is defined by the Standard in terms of interface, not implementation. The implementation of `vector`, for example, is made to be incredibly efficient by those who implement the standard. But there's nothing special about `vector` at a language level; it's easy for a programmer to define their own data structures that are as easy to use as ones in the standard library, because any user-defined class can be made to work with the same functions/operators as standard library types.

However, I disagree with Hoare on the matter of pointers/references. He seems to despise them, since they allow for arbitrary assignment to any store location whatsoever. In fact, he remarks that: "Their introduction into high level languages has been a step backward from which we may never recover." It might just be my limited perspective talking, but I can't imagine computer science *without* pointers/references. So much of optimal C++ code, for example, requires using references and move semantics to not constantly be copying data. And pointers give rise to a number of data structures and algorithms that would otherwise be much less efficient. References/pointers can be dangerous, yes, but I still see them as somewhat fundamental.

Exercise 1F-3

There are two special considerations that need to be made when it comes to the division operator:

First, division between integers can (in mathematics) create non-integer results. However, I don't particularly want to add non-integer numbers to IMP. So the division I implement will be akin to division between integers in most typed languages: it will result in an integer that is the floor of the division operation.

Second, dividing by zero is undefined. I feel, though, like all control paths should be defined by the rules (even if they result in error). But if I introduced a new possible value of Aexp, like **undef**, then I would need to add new rules for all of the different arithmetic and Boolean expressions to accommodate the cases where $\langle e, \sigma \rangle \Downarrow \textbf{undef}$. While doable, that would explode the complexity of IMP in a very unpleasant way. Therefore, I've decided to define that dividing by zero results in zero. This may be "wrong," but dividing by zero is an undefined operation anyway, so I think it's fine. The new rules I introduce are:

$$\frac{\langle e_2 = 0, \sigma \rangle \Downarrow \textbf{false} \quad \langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 / e_2, \sigma \rangle \Downarrow \lfloor n_1 / n_2 \rfloor} \qquad \frac{\langle e_2 = 0, \sigma \rangle \Downarrow \textbf{true}}{\langle e_1 / e_2, \sigma \rangle \Downarrow 0}$$

Exercise 1F-4

Our one new rule for the `let` command is:

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

Broadly, this rule indicates that the `let` command should run c on a state σ', where σ' is the result of running $x := e$ on σ. After running c on σ' to obtain σ'', the value of x in this state is replaced with whatever value it had before the call to `let`.

Exercise 1F-5

We don't need any additional contexts, since the `let` statement will always be matched with the context $H = \bullet$ based on our added redex. We modify

the redexes by adding:

$$r ::= \ldots \mid \texttt{let}\ x = e\ \texttt{in}\ c$$

And we add the local reduction rule:

$$\langle \texttt{let}\ x = e\ \texttt{in}\ c, \sigma \rangle \rightarrow \langle x := e; c; x := \sigma(x), \sigma \rangle$$

Obviously, when using this rule, $\sigma(x)$ is evaluated as whatever x's value is in the current state. This performs the same operations as were described in the previous problem; setting x to its new value/expression, running the command, and then setting x back to its original value.

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

- **0 pts** Correct