Exercise 2. During reading Hoare's *Hints On Programming Language Design*, one of the parts that I found the most interesting and attracting was the section "4. Comment Conventions". The section makes several points regarding designing comment conventions, among which I am in particular impressed by the following two points (corresponding to the first two points in the only list in that section):

1. large comment overheads should be avoided as they are "particularly discouraging to short comments";

2. comment conventions should be designed such that a comment is never limited to appear only at particular positions, especially when "it would sometimes be more relevant elsewhere".

The Item 1 is especially inspiring, indicating that comment conventions can potentially affect how people write comments, and even whether people write comments or not! As a result they potentially affect at least the readability of programs, which I did not realized in the past. The Item 2 aligns closely with my personal programming experience, that I am used to place comments in various places throughout a program, even in the middle of an expression e.g. for explaining immediately what is going on in a tricky calculation. To further exemplify, the comment convention is exactly among the language features that, not to offend, prevent me from being a fan of Python (with the indent-oriented syntax being another such feature by the way). There is only the `#...` syntax for end-of-line comments and there is no way to put comments in the middle of an expression, which violates Item 2 in the aforementioned list of desired properties of comment conventions — people often use the `'''...'''` structure for multiline comments, but that is essentially syntax for string literals and thus cannot be placed arbitrarily among expressions, and even if it could, the triple-apostrophes could be considered large overheads violating Item 1.

Among the ingenious and clearly elaborated points in the paper, there nevertheless exists some parts that I fail to appreciate. In the section "8. Variables", it is claimed that "the concept of reference, pointer, or indirect address" immediately damages the benefits of using separate variables to secure programs. However, I would consider references essential to efficient programs where it is critical that large data structures are able to update partially, instead of having a full copy at each update; e.g. we would definitely like to have efficient trees of various kinds, and, as a super familiar example, my first submission to the coding Exercise 1C this time copied the whole state hash table at `let` commands, and not surprisingly received `Timed out`. Hence as a result there must exist mechanisms for specifying the part in a (large) data structure that is shared before and after an update, which, as far as I could conceive, have the role equivalent to references. On the other hand, references might not necessarily be harmful as long as we seek to live with them (which, as discussed above, are necessary and inevitable as long as we seek efficiency), and without compromising security. E.g. it turns out that a straightforward way to "fix the bug" introduced by references is to use them in a "purely functional" manner, allowing only constant references (pointers that point to constant type instead of pointers that point constantly, i.e., using the C syntax, `const T *` or `T const *` instead of `T * const` for some type `T`). Since the

troubles are caused as references allow programs to write arbitrarily, limiting references to be read-only gets rid of the troubles completely at least in the mentioned aspect. In fact, there do exist *immutable* (and even *persistent*) data structures that perfectly do (more than) the desired jobs, securely and efficiently.

The discussions around references remind me of Hoare's another famous statement, often referred to as "billion-dollar mistake":

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. ... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

In my opinion this statement is often misinterpreted as we should not have `null` in programming languages, but actually we could barely live without `null` — mathematically `null` is the base case for lists, trees, etc., and using it enables programs to be recursive in a clean way, instead of being filled with corner cases, and thus easy to reason about. Personally the statement might be more about separating `null` from references in type systems, making `null` a type of its own, using *nullable* references when necessary, and optionally setting references to be non-nullable by default. In summary the discussions about both references and `null` (originate with Hoare and) demonstrate that for every feature of programming languages there are of course advantages in expressiveness while there might also be potential harm to program complexity and security; sometimes we might have to trade off, while sometimes it might turn out to be possible to achieve the best in both worlds, by figuring out and eliminating unnecessary use cases of the feature (e.g. non-constant references and unnecessary non-nullable references discussed so far).

Exercise 3. In order to extend the operational semantics of IMP with division operator, we could and it suffices to add the following new rule for the division operator:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1/e_2, \sigma\rangle \Downarrow n_1/n_2} \ .$$

Here the mathematical expression n_1/n_2 could stand for $\lfloor n_1/n_2 \rfloor$ or $\lceil n_1/n_2 \rceil$ or $\lceil n_1/n_2 \rfloor$ (either round half up, or round half down, round half towards zero, round half towards infinity, etc.; see `https://en.wikipedia.org/wiki/Rounding`), or even the rational division if IMP is further extended from integral data type to rational data type.

Exercise 4. Notice that the command `let` $x = e$ `in` c is equivalent to $x := e; c$ except that at the end of c the value of x should be recovered to its old value as if the whole command is not executed. Therefore the new rule for the `let` command could be:

$$\frac{\langle x := e; c, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{let}\ x = e\ \texttt{in}\ c, \sigma\rangle \Downarrow \sigma'[x := \sigma(x)]} \ .$$

Note that here referencing $\sigma(x)$ in $\sigma'[x := \sigma(x)]$ is valid as these are mathematical expressions. By composing with the rules for ";" and ":=", it is also equivalent[1] to have the following rule for the `let` command:

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

Exercise 5. Similar to the previous Exercise, we could aim at translating `let` $x = n$ `in` c to $x := n; c$ followed by a recovery of the value of x. This kind of translation is similar to what we did for the `while` command (when the loop guard is true). Analogous to the toolkit for the `while` command, the contexts, redexes, and reduction rules for the `let` command could be extended as follows:

$$\begin{aligned} H &::= \ldots \text{ (i.e. nothing to extend)} \ , \\ r &::= \cdots \mid \texttt{let}\ x = n\ \texttt{in}\ c \ , \\ \langle \texttt{let}\ x = n\ \texttt{in}\ c, \sigma\rangle &\to \langle x := n; c; x := \sigma(x), \sigma\rangle \ . \end{aligned}$$

Note that here referencing $\sigma(x)$ in $x := n; c; x := \sigma(x)$ is valid as the reduction rules are mathematical constructions. Moreover, by composing with the reduction rules for ";" and ":=", the new reduction rule can be optionally slightly simplified as:

$$\langle \texttt{let}\ x = n\ \texttt{in}\ c, \sigma\rangle \to \langle c; x := \sigma(x), \sigma[x := n]\rangle \ .$$

[1]Strictly speaking the second rule is only necessary compared with the first rule, while it could be considered sufficient as long as we only care about the inferences about the program and do not care which path we follow to reach the inferences.

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

- **0 pts** Correct