2 Exercise 1F-2. Language Design

There are many compelling "hints" Hoare presents in this paper, and on the whole I must say it really strikes me as prescient (or perhaps, we simply haven't made very much progress in language design since the '70s...). But the most compelling to me is his discussion of simplicity in language design. Over the last ten-or-so years, we've seen quite a lot of new languages (Go, Swift, Scala, Rust) crop up and achieve mainstream use, and this trend shows few signs of stopping. In a programming ecosystem that becomes increasingly heterogeneous, it becomes more important than ever to be able to quickly pick up and be productive in a new language. Hoare states that "the true craftsman is the one who thoroughly understands his tools". And from this, he asserts (correctly in my view) that simple language design that permits quick and easy learning is essential for a programmer to reach their peak effectiveness.

Perhaps the most prescient sub-point of this overall push for simplicity is Hoare's borderline dismissal of modularity. While it remains a common approach for dealing with language complexity to this day, I believe Hoare's criticisms ring true. Regardless of how organized and modular a language's featureset is, there will always be holes in the abstraction, where a programmer may need to "get into the weeds" to resolve an issue. And even with sophisticated static analysis and compiler support, as Hoare mentions, whatever compiler output the programmer sees is likely to be useless if they aren't familiar with the module in which they've unwittingly become entangled. The reason I find Hoare's point on modularity so compelling is that I believe, in the modern day, many new languages are making this same mistake. Rust, for instance, is an incredibly exciting language - safety and performance guaranteed by the compiler is an amazing sell. But in doing so, the language introduces considerable complexity. I've tried to learn Rust a number of times, and ultimately I managed to figure it out - but it was a painful process, and I still avoid the language for personal projects. I think a lot of this difficulty was actually orthogonal to Rust's core concepts of ownership and memory safety. The designers have hyper-emphasized modules (the famous crate system), such that the vast majority useful language functions are compartmentalized. Even core language features, like marking a type as copyable, are strictly named and modularized (in this case, `std::marker::Copy`). This extremely nominative approach to types and code organization has always been the main thing I find difficult when I'm trying to write useful code in Rust, and I believe it contributes strongly to Rust's reputation as a difficult language to learn. And it's a shame. Rust's mission of safety and performance is a good one, but it's impeded by its high-level language design. In contrast, a language like Google's Go - which has little in the way of innovation or interesting features - becomes wildly popular and famous for its simplicity and ease of use. It's possible to argue this is due to corporate backing, but I don't think this is the case - Google has *many* languages, but Go is far and away the most successful. It seems simplicity still wins out, even now.

I agree with most of the major points of this paper, honestly. It's hard to find anything significant that I disagree with. But one minor statement I find a little dated is the notion that expression syntax for non-arithmetic operations should always be visually distinct. For instance, Hoare specifically mentions syntax such as `.+` for adding two matrices, or `.append()` for adding two lists. In some senses this makes code more explicit, and also serves as a marker for a more expensive operation. But computers are much faster now, and in practice I find the added clarity is rarely necessary. I personally have experience writing shaders in GLSL, for example, which heavily rely on vectors and matrices and use the ordinary addition and multiplication operators for them. In a statically-typed language like GLSL, it's clearly unambiguous to the compiler which type of operation to perform based on the operand types. And to the reader, I find it's generally clear based on contextual type annotations and variable naming what operation is intended. In contrast, in languages like Java where no expression overloading is permitted, matrix operations become very verbose, and ultimately less graceful and readable than if the more ambiguous syntax had been chosen.

3 Exercise 1F-3. Simple Operational Semantics

Integer division is fraught with danger, and a lot of different languages have a lot of different approaches for it. In IMP, since there are no types other than int, and no exception system, we will use the following rules:

1. If any integer is divided by zero, the result will be zero. This grants totality to our division function, and this approach is used in several existing languages such as Coq and Pony.
2. If the quotient produced by the division is not an integer, it will be rounded towards $-\infty$.
3. Otherwise, the quotient will be the result of applying arithmetic division to the two operands.

We formalize these rules with the following inferences:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1/e_2, \sigma\rangle \Downarrow 0} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow n_2 \neq 0}{\langle e_1/e_2, \sigma\rangle \Downarrow \lfloor n_1/n_2 \rfloor}$$

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

The `let` command has some interesting and unique properties. We need to produce a binding for the new local variable that does not supplant an existing binding for the same name - that is, after the `let` command has concluded, we should restore any existing bindings for said variable. However, the body of the `let` can mutate existing bindings for other variables, so we can't simply restore the previous environment. We'll need to remove just the binding for the `let`'s new local variable. Let's denote an environment σ without the most-recent binding for variable x as $\sigma - x$.

We formalize this construct by extending $\langle c, \sigma\rangle \Downarrow \sigma'$ with the rule $\dfrac{\langle e, \sigma\rangle \Downarrow n_1 \qquad \langle c, \sigma[x := n_1]\rangle \Downarrow \sigma'}{\langle \text{let } x = e \text{ in } c, \sigma\rangle \Downarrow \sigma' - x}$

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

To describe the `let` construct in small-step semantics, we will extend the context H and redex grammar r with the following rules:

$\langle H\rangle$ ::= ...
| let x = H in c
| let () in H

$\langle r\rangle$ ::= ...
| let $x = n$ in c
| let () in *skip*

We thus define the reduction rules for the added redexes, using the same notation for removing a binding as in Exercise 1F-4:

$$\langle \text{let } x = n \text{ in } c,\ \sigma\rangle \to \langle \text{let () in } c,\ \sigma[x := n]\rangle$$
$$\langle \text{let () in } \mathit{skip},\ \sigma\rangle \to \langle \mathit{skip},\ \sigma - x\rangle$$

Peer Review ID: 63219454 — enter this when you fill out your peer evaluation via gradescope

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

- **0 pts** Correct