Exercise 1F-2. Language Design [5 points]

For:
In Section 2 of Hoare's writing, he makes the point that *a good programming language will encourage the programmer to write clear self-documenting code*, a point I strongly agree with. As I am still in the beginning stages of my programming career, I have had to learn many different programming languages throughout the course of my internships. Two languages stand out to me during my learning experience. PHP: I had a pretty miserable experience looking through and trying to understand how different pieces of PHP code functioned. Much of the syntax and behavior of the code was quite confusing to me and required extensive comments in the code to understand. Scala: the language itself was quite intuitive, and it was much easier for me to follow along with what was happening in different portions of the code, even without any comments. I was able to ramp up much faster working with the Scala codebase and was able to deliver meaningful contributions in much shorter timeframes. Much of a programmer's job is in maintaining older codebases, and this requires being able to easily read through, understand, and then updating the code. In languages where the code itself is not as self-documenting, there is a much higher burden on both on the original developper to maintain readibility and a much higher cost for any code maintenance/refactoring to occur.
Against:
I would argue against Hoare's requirement of simplicity, and instead argue for modularity, one of the design patterns he actually disputes in section 3.1. With a well-implemented system of modularity, a programmer will not need to understand the entirety of a language to effectively code in that language, while still allowing the language to contain modules that can support more complex behaviors. By having these modularized, more complex pieces, a language can allow more advanced programmers to develop cleaner and more efficient code without really negatively affecting the experience of more beginner programmers in the language. Furthermore, with modern IDEs and debugging tools it is much easier for a programmer working in a new language to avoid accidentally invoking language behaviors they did not intend to. At one point in my internship this past summer, I had to update some python code and allow support different functions with similar/same behaviors. Through the use of Python's function decorators, I was able to cleanly add this functionality to each of the functions without code duplication, in addition to allowing any new reader to know which functions in the code utilized this new functionality. Within Python, decorators are a clearly modularized piece of the language that allows more familiar programmers to better encapsulate their behavior without stepping on the toes of a beginner in the language.

Exercise 1F-3. Simple Operational Semantics [3 points]

The lefthand side division operator is division for the IMP language. The righthand side division operator is arithmetic division for integers, and we floor the value so we get appropriate integer division. Since division by 0 is undefined, there's no point in defining the value.

$$\frac{e_1, \sigma \Downarrow n_1 \qquad e_2, \sigma \Downarrow n_2}{< e_1 / e_2, \sigma > \Downarrow \lfloor n_1 / n_2 \rfloor} \quad \text{if } n_2 \neq 0$$

Exercise 1F-4. Language Feature Design, Large Step [10 points]

The goal is to create a state in which x is set to e and use that state to evaluate c. However, we need to be sure we can retrieve the value of x at which we stored it, so we've saved it into its own state variable. Thus, we end up with

$$\frac{< x := e, \sigma_0 > \Downarrow \sigma_1 \qquad < c, \sigma_1 > \Downarrow \sigma_2 \qquad < x := \sigma_0(x), \sigma_2 > \Downarrow \sigma_3}{< \text{let } x := e \text{ in } c, \sigma_0 > \Downarrow \sigma_3}$$

Exercise 1F-5. Language Feature Design, Small Step [10 points]

We want to be able to handle both the simple and the more complicated case. The simple case of our let statement occurs if the statement does not indeed change the x value at all. Thus, we have the reduction statement:

$$< \text{let } x := n \text{ in } c, \sigma > \rightarrow < c, \sigma > \quad \text{if } n = \sigma(x)$$

We need to also be able to handle reductions in the case where $n \neq \sigma$, so we end up with the following additional reduction statement:

$$< \text{let } x := n \text{ in } c, \sigma > \rightarrow < x := n; c; x := \sigma(x), \sigma > \quad \text{if } n \neq \sigma(x)$$

The redex comes to us for free from our reductions

$$\text{let } x := n \text{ in } c$$

Now, we need our context. In order to appropriately define what we can recurse on, we need to figure out what variables. Since c already has its own context and reduction rules as seen in lecture, we don't need to define anything else for it. Then, we focus on the *let* $x := e$. We cannot substitute out the entire $x := e$ expression as we need the value of e during our redex and reduction steps. Thus, we substitute

out the e for H, so that we can obtain the value we need and still allow for further recursion. Thus, we end up with our context

$$\text{let } x := H \text{ in } c$$

Now, putting all our pieces together, we end up with our final solution:

$$\begin{aligned}
&H ::= \ldots \mid \text{let } x := H \text{ in } c \\
&r ::= \ldots \mid \text{let } x := n \text{ in } c \\
&< \text{let } x := n \text{ in } c, \sigma > \rightarrow < c, \sigma > && \text{if } n = \sigma(x) \\
&< \text{let } x := n \text{ in } c, \sigma > \rightarrow < x := n; c; x := \sigma(x), \sigma > && \text{if } n \neq \sigma(x)
\end{aligned}$$

Exercise 1C. Language Feature Design, Coding

Submitted on Autograder

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

- **0 pts** Correct