2 Exercise 1F-2. Language Design

I found the commentary on documentation to be particularly relevant to my experience. Trying to understand a large code base, particularly one where you didn't write all of it, is a common task when using external tools or collaborating on a team. Hoare mentions the term "self-documenting code"(page 3), which I find to be an incredibly useful notion, both for language designers and for the writers of software. Writing additional commentary takes time, and has more subjectivity involved in its creation, as well as variance in how it is consumed. The english language does not enforce precise meaning in that same way as a programming language, nor is it forced to be linked with the behavior of the code it is describing. From a practical standpoint, external documentation requires extra effort to be maintained and updated by its writers, and additional effort for a reader to map that meaning back to the particular semantics of a section of code.

Simplicity in language design, as Hoare argues for in depth, can make arbitrary code written in a language easier to read, without additional effort from the writer. Such simplicity reduces the need for external documentation, and makes required documentation easier to understand. Increased readability further aids the programmer as they debugissues, and decreases the likelihood of writing errors due to complexities they haven't internalized.

3 Exercise 1F-3. Simple Operational Semantics

To implement a division operator, the abstract syntax of Aexp would be extended to include Aexp/Aexp. The semantics would be evaluated as the left expression being divided by the right expression. Because we only have integers within the IMP language, this division will be considered integer division within the Aexp sub language. To extend to floating point division, a new type would need to be added to this language. Similarly, division by zero can be handled arbitrarily, based upon the arithmetic definition of the division upon the two numbers. Formally, the operational semantics would be:

$$\frac{< e_1, \sigma >\Downarrow n_1 \quad < e_2, \sigma >\Downarrow n_2}{< e_2/e_2, \sigma >\Downarrow n_1/n_2}$$

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

The let operation has a notion of scoping the value of a variable, and thus must return it to its value prior to the beginning of the scope. Formally, this can be annotated as evaluating the enclosed command with the value assigned by the let, and then modifying the output state to reassign the variable to its previous state(and therefore, let does not increase the expressiveness of the language). Formally,

$$\frac{< x, \sigma >\Downarrow n \quad < x := e; c, \sigma >\Downarrow \sigma'}{< \text{let } x = e \text{ in } c, \sigma >\Downarrow \sigma'[x := n]}$$

Question assigned to the following page: 5

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

To extend our small step semantics with this let rule, we can use the redexes to explicitly append the assignment of the old value, and then verify that it is evaluated first in the context expressions. The redexes and contexts are thus extended with:

$$\begin{aligned} r ::= \ldots &| \langle \text{let } x = e \text{ in } c, \sigma \rangle \rightarrow \langle \text{ let } x = e \text{ in } c, x := x, \sigma \rangle \\ &| \langle \text{let } x = e \text{ in } c; x := n, \sigma \rangle \rightarrow \langle x := e; x; x := n, \sigma \rangle \\ H ::= \ldots &| \text{ let } x = e \text{ in } c; x := H \end{aligned}$$

As a worked example, here is the steps of the reduction from some initial state with $\sigma[x] = 10$:

$\langle Comm, State \rangle$	Redex	Context
$\langle$ let $x = e$ in $c, \sigma \rangle$	let x = e in c	•
$\langle$ let $x = e$ in $c; x := x, \sigma \rangle$	x	let x = e in c; x:= •
$\langle$ let $x = e$ in $c; x := 10, \sigma \rangle$	let x = e in c; x:=10	•
$\langle x = e; c; x := 10, \sigma \rangle$		

Which can then be evaluated using the previous rules of the language.