EXTENDS *Naturals*, *FiniteSets*

Imports standard modules that define operators of arithmetic on natural numbers and the *Cardinality* operator, where $Cardinality(S)$ is the number of elements in the set S, if S is finite.

CONSTANT *Value*

The set of all values that can be chosen.

VARIABLE *chosen*

The set of all values that have been chosen.

The type-correctness invariant asserting the "type" of the variable 'chosen'. It isn't part of the spec itself–that is, the formula describing the possible sequence of values that 'chosen' can have in a behavior correct behavior of the system, but is an invariance property that the spec should satisfy.

$$TypeOK \triangleq \begin{array}{l} \land\ chosen \subseteq Value \\ \land\ IsFiniteSet(chosen) \end{array}$$

Figure 1: A Beautifully Formatted Snippet from Leslie Lamport's Turing Award Winning PAXOS Code (yes this is real compilable code)

Exercise 1F-2. Language Design

Where I think Hoare Is Correct

Hoare describes that "A good programming language will encourage and assist the programmer to write clear self-documenting code, and even perhaps to develop and display a pleasant style of writing." This evokes thoughts of easily readable modern languages like Python. Python, being slower than many other languages has gained widespread popularity through usability and readability. Python programs are unusually easy to write and often read nearly like English. While I enjoy a Python project as much as anyone, in my experience the language which best captures the idea that Hoare is targeting here is Lamport's TLA+. This language is not so much a programming language as a mathematical protocol modeling language. TLA+ builds up mathematical models of the concept an engineer may want to write. Because the language is directly composed of math formulas, it is easily model checked. This captures another point Hoare covers. Hoare mentions that determining what you want your program to *do* is often as challenging as describing that to a compiler. TLA+ pulls this task to the front of the queue, where the program specification detailed in a ".tla" file and verified in the TLA+ IDE is also producible in very readable formats (fig 1). An implementation in some modern language can reference the verified and well formatted specification to achieve a correct and well designed algorithm born from the structure of thinking required by TLA+. TLA+ pulls together the idea of self documentation, and provides extremely effective aid to help an engineer think through the purpose of the algorithm they're designing.

Where I think Hoare Is Incorrect

Hoare describes the introduction of references and pointers as "a step backward from which we may never recover", and has famously taken responsibility for the introduction of the Null pointer. I note that Hoare doesn't distinguish between

so called "smart pointers" and actual pointers. I believe it's always a security risk to introduce pointers which may or may not get garbage collected later in the code. I agree that trying to track pointers throughout code causes many subtle bugs, and leads to a host of programming problems that could be avoided if pointers weren't used. All this still doesn't convince me of the argument that they represent a detriment for programming languages. Smart pointers have built-in security and automatic garbage collection which effectively leaves these points moot. Pointers were introduced for valid reasons and represent an idea that cannot be matched by other methods.

In some cases pointers offer algorithmic efficiency which cannot be reached through other methods. Less copying is required when using pointers, and structures like linked lists are most efficient when built using pointers. We often think of program efficiency in terms of how quickly we can run a program, time complexity, or how much memory it takes, space complexity. It's worth remembering that these metrics all eventually run on real machines. Every copy, iteration, etc, will cost some amount of energy. The world continues to move online, and our consumption transitions from physical commodities to cyber-products. While transport and manufacturing continue to lead the charge to damage the only livable planet we have, we should be careful to design programming languages which allow programmers to produce more efficient and therefore environmentally friendlier methods. Designing a language which can exploit the efficiencies of smart pointers, while avoiding the pitfalls of traditional pointers provides a middle ground Hoare seems to be overlooking here.

Exercise 1F-3. Simple Operational Semantics

We need to be careful about handling the case where the denominator is zero, in this case we should throw an error as division by zero is undefined. Because Aexps in IMP are integer in type, we also assume the mathematical division we use is integer division which is defined to be:

$$a \in \mathbb{Z}, b \in \mathbb{Z}$$

$$\frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor$$

$$\frac{< e_1, \sigma >\downarrow n_1 \quad < e_2, \sigma >\downarrow n_2}{< e_1/e_2, \sigma >\downarrow \lfloor n_1/n_2 \rfloor}$$

if $n1 \in \mathbb{Z}/\{0\}, n2 \in \mathbb{Z}/\{0\}$

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

Here we first find the previous value of x, in IMP if this variable is new it will be set to zero automatically. Next we find the value of e and augment σ to produce some σ' such that σ' is the result of c in sigma where x gets n2. Finally we execute a skip in σ' where x gets n1 resulting in the final σ'' where x is the same value as in σ, but any work done by c is preserved.

$$\frac{< x, \sigma >\downarrow n_1 \quad < e, \sigma >\downarrow n_2 \quad < c, \sigma[x := n_2] >\downarrow \sigma' \quad < skip, \sigma'[x := n_1] >\downarrow \sigma''}{< let \;\; x = e \;\; in \;\; c, \sigma >\downarrow \sigma''}$$

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

The small step implantation of

$$let\ x = e\ in\ c$$

Have H ::=

$$\bullet$$

(we can reduce the entire let statement using the rule I define two lines below.)
Add to r :: = x:

$$|\ let\ x = e\ in\ c$$

Add the following local reduction rule:

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

Notice that the right side of the reduction rule is a set of already existing rules. I show below how we can use the reduction I added to compute a concrete value.

A pseudo concrete execution of the rule might look like this. We have some initial state σ, which contains the original value of x as $\sigma(x)$. We then proceed to take single steps until we have σ', which is exactly sigma, but with x now set to the final value of e, I call this n. Next we run the command c, and produce σ''. Finally we set x = $\sigma(x)$ thus producing the third and final σ'''.

⟨ Comm, State ⟩	Redex •	Context
⟨ let x := e in c, σ⟩		
⟨ x := e; c; x := $\sigma(x)$, σ⟩	e	x := •; $c; x := \sigma(x)$
⟨ x := n; c; x := $\sigma(x)$, σ⟩	x := n	•; $c; x := \sigma(x)$
⟨c; x := $\sigma(x)$, σ' ⟩	c	•; $x := \sigma(x)$
⟨x := $\sigma(x)$, σ'' ⟩	$\sigma(x)$	x := •
⟨x := n, σ'' ⟩	x := n	•
⟨skip σ''' ⟩		

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

- **0 pts** Correct