Exercise 1F-2. Language Design [5 points].
A few years ago I would have said that ease of writing programs is more important than ease of reading them; but after spending a few summers working in the industry I fully agree that readibility is the far more important factor. It is painful to have to modify unreadible, convoluted code that you haven't written, and promoting readability is a good feature for a programming language to have. While Hoare makes a number of points worth of discussion and debate, I will focus on his arguments in favor of block/local scope and against references.

Hoare argues very convincingly in favor of block structure and local variable scope. Although block structure can sometimes cause unintuitive results when variable names are reused, scoping variables locally whereever possible does indeed strenghten the readability, maintability, and ease of coding a program. Anecdotally, I have tried using many global variables on several programming projects in the past, and it has *never* gone as easily as I anticipated: it is difficult to keep track of where global variables are and conceptually should be modified, whereas passing in parameters and declaring variables at the function level makes it much easier to get the program right and understand the logic. Aside from contexts like mutex-protected shared state across threads where local scope is unwiedly, local variables should be preferred over global (or even dynamically allocated) variables whose behavior is difficult to reason about. And even though most of us don't need to worry about our program exceeding main memory, the space savings from using local variables whose storage space can sometimes be substantial.

Hoare makes some incendiary remarks on references and pointers, claiming that "their introduction into high level languages has been a step backward from which we may never recover" (page 20). While any EECS 280 student could tell you that pointers and references are confusing and can be a major source of programmer error, references are actually a strength of modern programming languages in my opinion. Object-oriented programming makes heavy use of the Liskov Subsitution Principle that we can substitute objects of a child class for objects of a parent class, and base class pointers are generally how this is implemented in languages like C++. References can also provide tremendous space savings when passing a reference instead of a data structure filled with heavyweight object by value to a function. Using references correctly can indeed be challenging, but the reward of proper use is well worth the risk of misuse, making references a good feature to include in a programming language and not a hindrance.

Peer Review ID: 63273083 — enter this when you fill out your peer evaluation via gradescope

Exercise 1F-3. Simple Operational Semantics [3 points]. Consider the IMP language discussed in class, with the Aexp sub-language extended with a division operator. Explain what changes must be made to the operational semantics (big-step only). Write out formally any new rules of inference you introduce.

Solution: I introduce the following rule of inference for "IMPish" *integer* division:

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2 \quad \langle n_2 = 0, \sigma \rangle \Downarrow \text{false}}{\langle e_1/e_2, \sigma \rangle \Downarrow \lfloor n_1/n_2 \rfloor}$$

Informally, this rule:

1. Evaluates e_1 to n_1
2. Evaluates e_2 to n_2
3. Verifies that $n_2 \neq 0$
4. Performs integer division of n_1/n_2 by dividing n_1 and n_2 normally, then taking the floor. Alternatively, this can be viewed as finding the largest integer m such that $n_2 \cdot m \leq n_1$. (Note that $\lfloor\ \rfloor$ is defined as the regular floor operator.) So something like 3/2 would evaluate to $\lfloor 3/2 \rfloor = 1$.

The above rule deals with the case of correct behavior, but it does introduce some unintuitive behavior for the other Aexp sub-language rules/operational semantics (e.g. $e_1 + e_2$) in the case of illegal expressions like $(5/0) + 2$: they can have "side effects" and cause the program to never terminate. If we write $(5/0) + 2$, we have an $e_1 + e_2$, but with the above rule e_1 can't resolve the division operator and won't evaluate to any n_1, so we are stuck and cannot terminate. I would actually say that hanging/seg faulting/failing to resovle this illegal behavior case is fine, as IMP programs aren't guaranteed to termiante anyways (cf. while true do skip). An alterative, slightly more complicated resolution would be to introduce something to the effect of a FAIL state, then have an attempted division by zero (where $\langle n_2 = 0, \sigma \rangle \Downarrow \text{true}$) resolve to the FAIL state that should somehow cause the program to terminate with an error, but envisioning a "nicer" resolution for attempted illegal division in IMP is left as an exercise to the reader.

Peer Review ID: 63273083 — enter this when you fill out your peer evaluation via gradescope

Exercise 1F-4. Language Feature Design, Large Step [10 points]. Consider the IMP language with a new command construct "`let` $x = e$ `in` c". The informal semantics of this construct is that the Aexp e is evaluated and then a new local variable x is created with lexical scope c and initialized with the result of evaluating e. Then the command c is evaluated. We also extend IMP with a new command "`print` e" which evaluates the Aexp e and "displays the result" in some un-modeled manner but is otherwise similar to `skip`.

Extend the natural-style operational semantics judgment $\langle c, \sigma\rangle \Downarrow \sigma'$ with one new rule for dealing with the `let` command. Pay careful attention to the scope of the newly declared variable and to changes to other variables.

Solution: I introduce the `let` command for the above scenario:

$$\frac{\begin{array}{c}\langle e, \sigma\rangle \Downarrow n \qquad \langle x_{temp} := \sigma(x), \sigma\rangle \Downarrow \sigma[x_{temp} := \sigma(x)] \\ \langle x := e, \sigma[x_{temp} := \sigma(x)]\rangle \Downarrow \sigma[x_{temp} := \sigma(x), x := n] \qquad \langle c, \sigma[x_{temp} := \sigma(x), x := n]\rangle \Downarrow \sigma' \\ \langle x := x_{temp}, \sigma'\rangle \Downarrow \sigma'[x := x_{temp}] \qquad \langle x_{temp} := 0, \sigma'[x := x_{temp}]\rangle \Downarrow \sigma'[x := x_{temp}, x_{temp} := 0]\end{array}}{\langle \text{let } x \ = e \text{ in } c, \sigma\rangle \Downarrow \langle \sigma'[x := x_{temp}, x_{temp} := 0]\rangle}$$

Informally, this rule:

1. Evaluates e to n
2. Updates σ's state to contain x's old value (or 0, if x has not yet been initialized) so that old x can come back after new x goes out of scope.
3. Assigns the result of e, n, to x
4. Evaluates c with thie new x in scope
5. After c has resolved to σ', take new x out of scope by replacing its value with old x and zeroing out the temporary x value

While it requires a number of premises, this singular rule for `let` covers all cases in big-step fashion and appropriately deals with issues of scope.

Peer Review ID: 63273083 — enter this when you fill out your peer evaluation via gradescope

Exercise 1F-5. Language Feature Design, Small Step [10 points]. Extend the set of redexes, contexts and reduction rules for the contextual-style operational semantics that we discussed in class to account for the `let` command introduced above.

Solution: I introduce the following reduction rules, redexes, and contexts for `let`:

$H ::= \ldots | \cdot | if H \text{ then } c_1 \text{ else } c_2 | H; c | x := H | \text{let } H \text{ in } c$

$r ::= \ldots | \text{if true then } c_1 \text{ else } c_2; | \text{skip} ; c | x := n | \text{let } x \text{ in } c$

$\langle \text{let } x := n \text{ in } c, \sigma\rangle \rightarrow \langle \text{if } \sigma(x) = 0 \text{ then let } x := n \text{ in } c \text{ else } x_{temp} := \sigma(x); \text{let } x := n \text{ in } c, \sigma\rangle$
$\langle x_{temp} := \sigma(x); \text{let } x := n \text{ in } c, \sigma\rangle \rightarrow \langle \text{skip} ; \text{let } x := n \text{ in } c, \sigma[x_{temp} := \sigma(x)]\rangle$
$\langle \text{skip} ; \text{let } x := n \text{ in } c, \sigma\rangle \rightarrow \langle \text{let } x := n \text{ in } c, \sigma\rangle$
$\langle \text{let } x := n \text{ in } c, \sigma\rangle \rightarrow \langle \text{let skip in } c, \sigma[x := n]\rangle$
$\langle \text{let skip in } c, \sigma[x := n]\rangle \rightarrow \langle \text{skip} ; c, \sigma[x := n]\rangle$
$\langle \text{skip} ; c, \sigma[x := n]\rangle \rightarrow \langle c, \sigma[x := n]\rangle$

Peer Review ID: 63273083 — 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

Peer Review ID: 63273083 — enter this when you fill out your peer evaluation via gradescope