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

One thing Hoare talks about in Hints on Programming Language Design is how important it is for a programming language to be simple. He says it should be easy to learn in just a few days. I've felt this myself when I started learning Python. Its clean and simple design made it quick to pick up, and I could focus on solving problems instead of struggling to understand the language. This really showed me how much simplicity helps.

Hoare also mentions that too many runtime checks, like extra debugging or security checks, can make things more complicated than they need to be. I see his point, but in my experience, these checks are really helpful. For example, when using Java, its strict rules and type checks have caught mistakes early and saved me from bigger problems later. So while these checks might feel annoying during development, they're super useful in keeping programs safe and reliable.

To sum up, I agree with Hoare that simplicity in programming languages is important and makes a big difference. But I think he's wrong about runtime checks—they're essential for catching bugs and ensuring reliability, especially in larger, more complex projects. Without them, small errors during development can turn into major problems in production and thus would require more cost.

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

To extend the IMP language with a division operator (/), we need to modify the big-step operational semantics to handle division in the Aexp sub-language. The changes involve defining a new rule for evaluating division expressions and considering edge cases like division by zero.

1. Adding a New Rule for Division: We extend the big-step semantics to include a rule for division. This rule specifies how an expression of the form a1 / a2 is evaluated:

$$\frac{\langle a1, \sigma\rangle \Downarrow n1 \quad \langle a2, \sigma\rangle \Downarrow n2 \quad n2 \neq 0}{\langle a1/a2, \sigma\rangle \Downarrow n1/n2}$$

   - This rule says that if a1 evaluates to n1 and a2 evaluates to n2, and n2 is not zero, then a1 / a2 evaluates to n1 / n2.

2. Handling Division by Zero: To account for division by zero, we add a rule that handles this case as undefined:

$$\frac{\langle a1, \sigma\rangle \Downarrow n1 \quad \langle a2, \sigma\rangle \Downarrow 0}{\langle a1/a2, \sigma\rangle \text{ is undefined}}$$

   - This rule specifies that if a2 evaluates to 0, the result of a1 / a2 is undefined, as division by zero is not allowed.

To extend the IMP language with the new command constructs let x = e in c and print e, we need to update the natural-style operational semantics to handle the new behaviors of these commands. Below, I will define the changes needed to the operational semantics.

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

Questions assigned to the following page: 4 and 5

1. **Rule for let x = e in c**:

The let x = e in c construct creates a new local variable x, initializes it with the value of e, and then evaluates the command c in the context where x is available. To define this in the operational semantics, we need the following rule:

$$\frac{\langle e, \sigma \rangle \Downarrow n \quad \langle c, \sigma[x \mapsto n] \rangle \Downarrow \sigma'}{\langle \text{let } x = e \text{ in } c, \sigma \rangle \Downarrow \sigma'}$$

- ⟨e, σ⟩ ⇓ n: First, we evaluate the expression e in the current state σ, which gives the value n.
- ⟨c, σ[x \mapsto n]⟩ ⇓ σ': We then update the state σ to include a new binding x = n (i.e., the new variable x is initialized to n). After this, we evaluate the command c in the updated state σ[x \mapsto n].
- The final result is the state σ' after evaluating c, with x available only within the scope of c.

This rule makes sure that the variable x is only available in the scope of c and does not affect the state outside of c.

2. **Rule for print e**:

The print e command evaluates the expression e and displays its result, but does not modify the state. It behaves similarly to skip. The operational semantics for print e is given by the following rule:

$$\frac{\langle e, \sigma \rangle \Downarrow n}{\langle \text{print } e, \sigma \rangle \Downarrow \sigma}$$

- ⟨e, σ⟩ ⇓ n: We evaluate the expression e in the state σ, which results in the value n.
- The state σ remains unchanged because print only displays the result and doesn't alter the state.

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

From what was given in the class, the set of redexes, contexts and reduction rules can be extended as follows:

Step	Redex	Context	Next Step
1	x := 1	[x := 0, y := 0]	↑skip, [x := 1, y := 0]↓
2	y := 2	[x := 1, y := 0]	↑skip, [x := 1, y := 2]↓
3	let x = 3 in ...	[x := 1, y := 2]	↑print x; print y; x := 4; y := 5, [x := 3, y := 2]↓
4	print x	[x := 3, y := 2]	↑skip, [x := 3, y := 2]↓ (prints "3")
5	print y	[x := 3, y := 2]	↑skip, [x := 3, y := 2]↓ (prints "2")

Question assigned to the following page: 5

6	| x := 4	| [x := 3, y := 2]	| ↑skip, [x := 4, y := 2]↓
7	| y := 5	| [x := 4, y := 2]	| ↑skip, [x := 4, y := 5]↓
8	| print x	| [x := 1, y := 5]	| ↑skip, [x := 1, y := 5]↓ (prints "1")
9	| print y	| [x := 1, y := 5]	| ↑skip, [x := 1, y := 5]↓ (prints "5")

Detailed Explanation, step by step is as below:

Step 1 (x := 1):

- Redex: x := 1
- Context: [x := 0, y := 0] (initial state)
- Next Step: Updates x to 1 → [x := 1, y := 0]

Step 2 (y := 2):

- Redex: y := 2
- Context: [x := 1, y := 0]
- Next Step: Updates y to 2 → [x := 1, y := 2]

Step 3 (let x = 3 in ...):

- Redex: let x = 3 in ...
- Context: [x := 1, y := 2]
- Next Step: Creates a new scope where x = 3 → [x := 3, y := 2]

Step 4 (print x):

- Redex: print x
- Context: [x := 3, y := 2]
- Next Step: Prints 3, no state change → [x := 3, y := 2]

Step 5 (print y):

- Redex: print y
- Context: [x := 3, y := 2]
- Next Step: Prints 2, no state change → [x := 3, y := 2]

Step 6 (x := 4):

- Redex: x := 4
- Context: [x := 3, y := 2]
- Next Step: Updates x to 4 in the local scope → [x := 4, y := 2]

Step 7 (y := 5):

- Redex: y := 5

Question assigned to the following page: 5

- Context: [x := 4, y := 2]
- Next Step: Updates y to 5 → [x := 4, y := 5]

Step 8 (print x):

- Redex: print x
- Context: [x := 1, y := 5] (local scope ends, restores outer scope)
- Next Step: Prints 1, no state change → [x := 1, y := 5]

Step 9 (print y):

- Redex: print y
- Context: [x := 1, y := 5]
- Next Step: Prints 5, no state change → [x := 1, y := 5]