Exercise 1F-2. Language Design

Hoare's advice on **simplicity** matches my experience with Python and Java. Python is much easier for me to use because its syntax is simple and clear, which helps me focus on solving problems instead of worrying about complicated language rules. On the other hand, I found Java to be more complex because it has more rules and requires more code for even simple tasks, like handling errors. This makes programming slower and harder to follow, which goes against Hoare's idea of keeping things simple.

However, I also found that Hoare's advice on **clarity** isn't always perfect. Java's detailed syntax is clear in some ways, but it can make things harder to understand, especially when writing basic operations. For example, in Java, I have to write extra code to do simple things that Python handles more easily. This shows that too many rules or steps can actually make a language less clear and harder to use.

On the other hand, Hoare's point about **safety** is something I agree with, especially from my experience using C#. The **type system** in C# ensures that data types are used correctly, preventing errors that could otherwise occur, such as accidentally mixing up a number with a string. In languages like C# and Java, the type system is **strong and static**, which means you have to declare the type of each variable, and the language checks the types before the program runs. This catches errors early, making your code safer and more reliable. In contrast, **Python** uses a **dynamic type system**, which is more flexible but means that type errors might not be caught until the program is running. So, while Python is simpler and faster for development, I think C#'s stricter type system makes it safer in the long run, even if it adds some complexity.

Question assigned to the following page: 3

Exercise 1F-3. Simple Operational Semantics

To extend the `IMP` language with a division operator, we need to make the following changes:

1. The production rule for arithmetic expressions (`Aexp`) is updated to include division:

$$e ::= \cdots \mid e_1/e_2 \quad \text{for } e_1, e_2 \in \texttt{Aexp}$$

2. New rules of inference for division The operational semantics of division requires the addition of a new inference rule. Division by zero is not allowed.

Standard Case (Division by a Non-Zero Number)

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad n_2 \neq 0}{\langle e_1/e_2, \sigma\rangle \Downarrow n_1/n_2}$$

- e_1 and e_2 are evaluated under the state σ, producing n_1 and n_2.
- The condition $n_2 \neq 0$ ensures division by zero is not allowed.
- The result is the quotient n_1/n_2.

Error Case (Division by Zero)

If $n_2 = 0$, the semantics do not define a result, and the evaluation is invalid. This situation must be handled separately to prevent undefined behavior.

Question assigned to the following page: 4

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

To extend the operational semantics of the `IMP` language to handle the new constructs `let x = e in c` and `print e`, we define the following rules of inference.

New Constructs and Informal Semantics

1. **`let x = e in c`:** - The arithmetic expression e is evaluated in the current state σ to obtain a value n. - A new variable x is introduced with lexical scope restricted to c, and it is initialized to n. - The command c is then executed in the updated state, and the original state σ is restored upon completion.

2. **`print e`:** - The arithmetic expression e is evaluated in the current state σ to obtain a value n. - The value n is "displayed" (un-modeled in the semantics), but the state σ remains unchanged.

Formal Rules of Inference

We extend the natural-style operational semantics judgment $\langle c, \sigma \rangle \Downarrow \sigma'$ with the following rules:

Rule for `let x = e in c`

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

- e is evaluated under the state σ to produce a value n. - The command c is then executed under an updated state $\sigma[x := n]$, where x maps to n. - Once c completes, the scope of x ends, and the original state σ is restored.

Rule for `print e`

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

- e is evaluated under the state σ to produce a value n. - The value n is displayed (un-modeled in the semantics), and the state σ remains unchanged.

The given program execution is as follows:

- $x := 1$: Updates $\sigma(x) = 1$.
- $y := 2$: Updates $\sigma(y) = 2$.
- `let x = 3 in` $\{\ldots\}$:
  - $x = 3$: Introduces a local variable x with value 3.

Question assigned to the following page: 4

  - `print x`: Displays 3.
  - `print y`: Displays 2 (outer y remains 2).
  - $x := 4$: Updates the local x to 4.
  - $y := 5$: Updates the outer y to 5.

  Upon exiting the `let` scope, the local x is discarded, and $\sigma(x)$ reverts to 1.

- `print x`: Displays 1 (restored value of x).
- `print y`: Displays 5 (updated value of y).

The program displays: `3 2 1 5`.

Question assigned to the following page: 5

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

Step	Redex	Context	Next Step
1	$x := 1$	$[x := 0, y := 0]$	$\langle \text{skip}, [x := 1, y := 0] \rangle$
2	$y := 2$	$[x := 1, y := 0]$	$\langle \text{skip}, [x := 1, y := 2] \rangle$
3	let $x = 3$ in $\ldots$	$[x := 1, y := 2]$	$\langle \text{print } x; \text{print } y; x := 4; y := 5, [x := 3, y := 2] \rangle$
4	print x	$[x := 3, y := 2]$	$\langle \text{skip}, [x := 3, y := 2] \rangle$ (prints "3")
5	print y	$[x := 3, y := 2]$	$\langle \text{skip}, [x := 3, y := 2] \rangle$ (prints "2")
6	$x := 4$	$[x := 3, y := 2]$	$\langle \text{skip}, [x := 4, y := 2] \rangle$
7	$y := 5$	$[x := 4, y := 2]$	$\langle \text{skip}, [x := 4, y := 5] \rangle$
8	print x	$[x := 1, y := 5]$	$\langle \text{skip}, [x := 1, y := 5] \rangle$ (prints "1")
9	print y	$[x := 1, y := 5]$	$\langle \text{skip}, [x := 1, y := 5] \rangle$ (prints "5")