Exercise 2F-2. Mathematical Induction [5 points].

Proof: Let F be the set of all flowers and let $smells(f)$ be the smell of the flower $f \in F$.
(The range of smells is not so important, but we'll assume that it admits equality.) We'll also assume that F is countable.
Let the property $P(n)$ mean that all subsets of F of size at most n contain flowers that smell the same.

$$P(n) = \forall X \in P(F), |X| \leq n \Rightarrow (\forall f, f' \in X, smells(f) = smells(f'))$$

One way to formulate the statement to prove is $\forall n \geq 1, P(n)$. We'll prove this by induction on n, as follows:

Base Case: $n = 1$.
Obviously, all singleton sets of flowers contain flowers that smell the same (by the definition of $P(n)$).

Base Case: $n = 1$.
Obviously, all singleton sets of flowers contain flowers that smell the same (by the definition of $P(n)$).

Induction Step:
Let n be arbitrary and assume that all subsets of F of size at most n contain flowers that smell the same.
We will prove that the same thing holds for all subsets of size at most $n + 1$. Pick an arbitrary set X such that $|X| = n + 1$.
Pick two distinct flowers $f, f' \in X$ and let's show that $smells(f) = smells(f')$.
Let $Y = X - \{f\}$ and $Y' = X - \{f'\}$.
Obviously, Y and Y' are sets of size at most n, so the induction hypothesis holds for both of them.
Pick any arbitrary $x \in Y \cap Y'$.
Obviously, $x \neq f$ **and** $x \neq f'$.

$$smells(f') = smells(x) \quad \text{(from the induction hypothesis on } Y\text{)}$$

$$smells(f) = smells(x) \quad \text{(from the induction hypothesis on } Y'\text{)}$$

Hence, $smells(f) = smells(f')$**, which proves the inductive step, and the theorem.**

Peer Review ID: 306391518 — enter this when you fill out your peer evaluation via gradescope

Question assigned to the following page: 3

Exercise 2F-3. While Induction [10 points].

$$\langle \text{while } b \text{ do } x := x + 2, \sigma \rangle \Downarrow \sigma'$$

where $\sigma(x)$ is even, and we need to show that $\sigma'(x)$ remains even.

The two key cases for the execution of a while-loop are:

1. Best Case (Termination without loop execution)

   * If b is false in σ, the loop does not execute and σ′=σ
   - Since σ(x) is even by assumption, σ′(x)=σ(x) is also even. The property holds.

2. Inductive Step (Loop Executes at Least Once)

   - If b is true in σ, the loop does executes atleast once.
   - The first iteration updates x:=x+2, and we obtain a new state σ1 where:

$$\sigma_1(x) = \sigma(x) + 2.$$

- Since the sum of two even numbers is even, $\sigma_1(x)$ remains even.
- The remaining execution follows the derivation:

$$\langle \text{while } b \text{ do } x := x + 2, \sigma_1 \rangle \Downarrow \sigma'.$$

- By the **inductive hypothesis**, $\sigma'(x)$ remains even.

By the base case, the property holds when the condition is initially false. By the inductive step, assuming the property holds for states where x remains even after loop body execution, it must hold for the current state as well.

Consequently, for any σ such that σ(x) is even, σ′(x) will also be even after executing the while-loop. This completes the structural induction proof.

Question assigned to the following page: 4

Exercise 2F-4. Language Features, Large-Step [12 points].

Large-step operational semantics inference rules **for the three new commands:** throw e, try c1 catch x c2, **and** after c1 finally c2 are as follows:

Rules for throw e:

1. Throwing an exception:

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

This rule states that evaluating throw e results in an exceptional termination with value
n, where n is the evaluation of the expression e

Rules for try c1 catch x c2

2. Normal execution of c1 (no exception thrown)

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \text{try } c_1 \text{ catch } x \; c_2, \sigma \rangle \Downarrow \sigma'}$$

If executing c1 completes normally without an exception, the entire try-catch block terminates normally with the final state σ

3. Handling an exception in catch

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \text{ exc } n \quad \langle c_2, \sigma'[x \mapsto n] \rangle \Downarrow t}{\langle \text{try } c_1 \text{ catch } x \; c_2, \sigma \rangle \Downarrow t}$$

If c_1 throws an exception n, the value of x is set to n in the state σ', and then c_2 is executed. The final result of the entire block is the result of executing c_2.

Rules for after c1 finally c2:

4. Normal execution of c1 followed by c2

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \quad \langle c_2, \sigma' \rangle \Downarrow t}{\langle \text{after } c_1 \text{ finally } c_2, \sigma \rangle \Downarrow t}$$

If c_1 executes normally and results in state σ', then c_2 is executed. The final termination result of the entire block is the result of c_2.

Peer Review ID: 306391518 — enter this when you fill out your peer evaluation via gradescope

Questions assigned to the following page: 4 and 5

5. C1 raises an exception, but c2 executes normally (exception rethrown)

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \text{ exc } e_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \text{ exc } e_1}$$

If c_1 throws an exception e_1, but c_2 executes normally, the final termination keeps the exception e_1 (it is rethrown after executing the `finally` block).

6. C1 raises an exception, and c2 overrides it with a new exception

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \text{ exc } e_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma'' \text{ exc } e_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \text{ exc } e_2}$$

If c_1 throws an exception e_1, and c_2 also throws an exception e_2, the final termination of the block is with e_2, overriding e_1.

In conclusion:

- The throw rule directly raises an exception with the value obtained from the expression.
- The try-catch rules handle two scenarios: if the first command c1 completes normally, the entire block completes normally. If c1 raises an exception, it is caught. The value is assigned to x, and c2 is executed.
- For the finally command, if c1 executes without exceptions, then c2 executes next, with the termination result depending entirely on c2. If c1 throws an exception, c2 is still executed. Depending on whether c2 raises a new exception or completes normally, the final result is determined.

Exercise 2F-5. Language Features, Analysis [6 points].

Using small-step semantics for exceptions is more natural and elegant because it provides fine-grained control over execution, making it easier to track how exceptions propagate through a program. Since exceptions can occur at any point during execution, small-step semantics captures this process step by step, rather than just showing the final outcome. This is especially useful for modeling non-termination, as it allows us to see if a loop runs indefinitely before an exception occurs. Additionally, small-step semantics is more modular and extendable, making it easier to introduce features like stack traces or nested exceptions without overhauling the entire system. It also closely mirrors how real-world languages like Java and Python handle exceptions—execution doesn't happen in one big leap but as a series of incremental steps.

On the other hand, large-step semantics has some advantages, such as being simpler and more readable for small programs where we only care about the final result. However, this approach falls short when exceptions interrupt execution before completion, as it doesn't show the intermediate steps leading to an error. Because exception handling is inherently about managing disruptions mid-execution, small-step semantics provides a clearer and more realistic representation of how exceptions behave in practice. Therefore, small-step semantics is the more natural choice for modeling exceptions in IMP.