Exercise 2F-2. Mathematical Induction [5 points]. *Proof:* Let F be the set of all flowers and let $\mathsf{smells}(f)$ 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.

$$P(n) \overset{\text{def}}{=} \forall X \in \mathcal{P}(F).\ |X| \leq n \implies (\forall f, f' \in X.\ \mathsf{smells}(f) = \mathsf{smells}(f'))$$

(the notation $|X|$ denotes the number of elements of X)

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 $P(n)$).

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 $\mathsf{smells}(f) = \mathsf{smells}(f')$. 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 $x \neq f'$. We have that $\mathsf{smells}(f') = \mathsf{smells}(x)$ (from the induction hypothesis on Y) and $\mathsf{smells}(f) = \mathsf{smells}(x)$ (from the induction hypothesis on Y'). Hence $\mathsf{smells}(f) = \mathsf{smells}(f')$, which proves the inductive step, and the theorem.

Here, the proof makes an incorrect assumption in the inductive step while picking the two flowers from convenient sets Y and Y'. The problem is that the intersection of Y and Y' is not necessarily non-empty, and we observe this when trying to show that $P(1) \implies P(2)$, which fails. It is incorrect to assume that some common flower exists in both Y and Y'. So, we cannot apply the inductive hypothesis to these potentially non-existent flowers to complete the proof argument.

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

Theorem. *For any BExp b and any initial state σ such that $\sigma(x)$ is even, if*

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

then $\sigma'(x)$ is even.

Proof. We will prove this property by performing *structural induction* on the derivation tree (based on the IMP large-step semantics rules).

There are two big-step rules for a `while` command.

WHILE-FALSE

$$\frac{\langle b,\sigma\rangle \Downarrow \textsf{false}}{\langle \textsf{while}\ b\ \textsf{do}\ c,\sigma\rangle \Downarrow \sigma}$$

WHILE-TRUE

$$\frac{\langle b,\sigma\rangle \Downarrow \textsf{true} \qquad \langle c; \textsf{while}\ b\ \textsf{do}\ c,\sigma\rangle \Downarrow \sigma'}{\langle \textsf{while}\ b\ \textsf{do}\ c,\sigma\rangle \Downarrow \sigma'}$$

So, we know the last rule could have either been the true or false case of the while command.

1. While-False Case (Base case): Here, b evaluates to false according to the premises, so the loop terminates immediately and hence, $\sigma'(x)$ is also even.

2. While-True Case (Inductive case): Here, b evaluates to true, and we run the loop once more recursively. Given, $\sigma(x)$ is even. So, one the leaves of the proof tree becomes $\langle x := x + 2; \textsf{while}\ b\ do\ x := x + 2, \sigma\rangle \Downarrow\ \sigma'$.

   Using the large-step rule for seq, first we run the set command ($\langle x := x + 2, \sigma\rangle \Downarrow \sigma_1$) which results in new memory state σ_1, where x is again even. Now, with this new σ_1, we make another recursive loop call ($\langle \textsf{while}\ b\ do\ (x := x + 2), \sigma_1\rangle \Downarrow\ \sigma'$), and now this invokes the inductive hypothesis that if a loop terminates from a state where x is even, then the final state will also have x even (re-using sub derivation within the proof tree).

The cases are exhaustive by definition, and by using structural induction on the derivation tree, we have proved the property about the while command.

□

Exercise 2F-4. Language Features, Large-Step [12 points]. We will extend the large-step operational semantics with 6 rules by adding 1, 2, and 3 rules for the throw, try − catch, and after − finally constructs, respectively.

Throw
$$\frac{\langle e\ ,\sigma\rangle \Downarrow n}{\langle \mathsf{throw}\ e, \sigma\rangle \Downarrow \langle \sigma'\ \mathsf{exc}\ n\rangle}$$

Try-Catch Normal
$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2, \sigma\rangle \Downarrow \sigma'}$$

Try-Catch Exception
$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n \qquad \langle c_2, \sigma'[x \mapsto n]\rangle \Downarrow t}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2\rangle \Downarrow t}$$

Finally-After Normal
$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \qquad \langle c_2, \sigma'\rangle \Downarrow t}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow t}$$

Finally-After Rethrow
$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma\ \mathsf{exc}\ n \qquad \langle c_2, \sigma\rangle \Downarrow \sigma''}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow \sigma''\mathsf{exc}\ n}$$

Finally-After Override
$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma\ \mathsf{exc}\ n \qquad \langle c_2, \sigma'\rangle \Downarrow \sigma''\ \mathsf{exc}\ n'}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow \sigma''\ \mathsf{exc}\ n'}$$

Exercise 2F-4. Language Features, Analysis [6 points]. I believe that describing the operational semantics for exceptions in IMP would have been more *natural* given the nature of the abrupt control flow changes that exceptions provide. I think small-step semantics are more well-suited to model such changes in an incremental and compositional fashion, where each rule precisely defines the next state, leading to simpler and more uniform rules. Instead, we can define exceptions as a sequence of small steps and have a matching handler at the top level, which gets triggers for its respective error cases.

Modeling such behavior feels rather *cumbersome* via large-step as we have to hard code rules for multiple scenarios (e.g., three rules for $\mathsf{finally} - \mathsf{after}$). I felt that this method yielded complicated and repetitive rules to model one style of exceptional behavior.