Exercise 2F-2. Mathematical Induction [5 points]. Find the flaw in the following inductive proof that "All flowers smell the same". Please indicate exactly which sentences are wrong in the proof via highlighting or underlining.

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.

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

(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.

(One indication that the proof might be wrong is the large number of occurrences of the word "obviously" :-))

Solution: The flaw is in the highlighted sentence above. Consider the case where n=1, and the set $X = \{f, f'\}$ with $|X| = 2$. Then, subset $Y = \{f'\}$ and $Y' = \{f\}$. Then $Y \cap Y'$ is the empty set, so you cannot pick an arbitrary $x \in Y \cap Y'$. This invalidates the inductive step for $n = 1 \to n = 2$.

Peer Review ID: 306029856 — 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]. Prove by induction the following statement about the operational semantics:

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. Make sure you state what you induct on, what the base case is and what the inductive cases are. Show representative cases among the latter. Do not do a proof by mathematical induction!

Solution:
Base Case: b evaluates to false (loop does not execute).

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

Since the state returned σ' is the same as the original state σ, and we know that $\sigma(x)$ is even, then $\sigma'(x) = \sigma(x)$, which is even.

Inductive Case: b evaluates to true (loop executes).

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

For our induction hypothesis, assume that for $\langle \text{while } b \text{ do } x := x + 2, \sigma_1 \rangle \Downarrow \sigma'$, σ' is even. Since $\sigma(x)$ is even, then $\sigma(x)+2$ is even, thus $\sigma_1(x)$ is even (after the loop body executes). The loop is then executed again from the state σ_1. By the induction hypothesis, since $x + 2$ will still be even, the new state will maintain the property that x is even. Thus, $\sigma'(x)$ is even when the loop finishes.

Question assigned to the following page: 4

Exercise 2F-4. Language Features, Large-Step [12 points]. We extend IMP with a notion of integer-valued *exceptions* (or *run-time errors*), as in Java, ML or C#. We introduce a new type T to represent command terminations, which can either be normal or exceptional (with an exception value $n \in \mathbb{Z}$):

$$\begin{array}{rcll} T & ::= & \sigma & \text{“normal termination”} \\ & | & \sigma \ \texttt{exc} \ n & \text{“exceptional termination”} \end{array}$$

We use t to range over possible terminations T. We then redefine our operational semantics judgment:

$$\langle c, \sigma \rangle \Downarrow T$$

The interpretation of

$$\langle c, \sigma \rangle \Downarrow \sigma' \ \texttt{exc} \ n$$

is that command c terminated abruptly by throwing an exception with value $n \in \mathbb{Z}$ at a point in c's execution when the state was σ'. We only model one type of exception, but every exception has an integer “argument” n (or “payload” or “value”) that is set when the exception is thrown and available when the exception is caught.

Note that our previous command rules must be updated to account for exceptions, as in:

$$[\mathsf{seq1}]\langle c_1; c_2, \sigma \rangle \Downarrow \sigma' \ \texttt{exc} \ n \langle c_1, \sigma \rangle \Downarrow \sigma' \ \texttt{exc} \ n \qquad [\mathsf{seq2}]\langle c_1; c_2, \sigma \rangle \Downarrow t \langle c_1, \sigma \rangle \Downarrow \sigma' \langle c_2, \sigma' \rangle \Downarrow t$$

We also introduce three additional commands:

$$\begin{array}{l} \mathsf{throw} \ e \\ \mathsf{try} \ c_1 \ \mathsf{catch} \ x \ c_2 \\ \mathsf{after} \ c_1 \ \mathsf{finally} \ c_2 \end{array}$$

- The throw e command raises an exception with argument e.
- The try command executes c_1. If c_1 terminates normally (i.e., without an uncaught exception), the try command also terminates normally. If c_1 raises an exception with value e, the variable $x \in L$ is assigned the value e and then c_2 is executed.
- The finally command executes c_1. If c_1 terminates normally, the finally command terminates by executing c_2. If instead c_1 raises an exception with value e_1, then c_2 is executed:
  - If c_2 terminates normally, the finally command terminates by throwing an exception with value e_1. (That is, the original exception e_1 is re-thrown at the end of the finally block, as in Java.)
  - If c_2 throws an exception with value e_2, the finally command terminates by throwing an exception with value e_2. (That is, the new exception e_2 overrides the original exception e_1, also as in Java.)

Question assigned to the following page: 4

These constructs are intended to have the standard exception semantics from languages like Java, C# or OCaml — except that the catch block merely assigns to x, it does not bind it to a local scope. So unlike Java, our catch does not behave like a let. We thus expect:

```
x := 0 ;
{ try
    if x <= 5 then throw 33 else throw 55
  catch x
    print x } ;
while true do {
  x := x - 15 ;
  print x ;
  if x <= 0 then throw (x*2) else skip
}
```

to output "`33 18 3 -12`" and then terminate with an uncaught exception with value `-24`.

Give the large-step operational semantics inference rules (using our new judgment) for the three new commands presented here. You should present six (6) new rules total.

Solution:

- **throw**:
  We introduce the rule to throw an exception and result in an exceptional termination:

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

- **try-catch**:
  First, we introduce the rule where c_1 is executed and results in a normal termination:

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

  Next, we introduce the rule where c_1 results in an exceptional termination:

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

- **finally**:
  First, we introduce the rule where both c_1 and c_2 result in normal terminations:

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

Question assigned to the following page: 4

Next, we introduce the rule where c_1 results in an exceptional termination, but c_2 results in a normal termination:

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

Finally, we introduce the rule where both c_1 and c_2 result in exceptional terminations:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \text{ exc } n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma'' \text{ exc } n_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \text{ exc } n_2}$$

Question assigned to the following page: 5

Exercise 2F-4. Language Features, Analysis [6 points]. Argue for or against the claim that it would be more natural to describe "IMP with exceptions" using small-step contextual semantics. You may use "simpler" or "more elegant" instead of "more natural" if you prefer. Do not exceed two paragraphs (one should be sufficient). Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter.

Solution: I believe that it would be more natural to describe "IMP with exceptions" using small-step contextual semantics. Exceptions can occur at any point in the program, and we want to be able to keep track of those intermediate states exactly where the exception occurred. Large-step semantics focuses more on the overall return state, while small-step semantics allows for more granularity of individual steps in the program's execution. Thus, using small-step semantics would make it simpler to represent the exact state an exception is thrown and how it branches off to different parts of the program, especially when there are more complex nested try-catch or finally statements.