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" :-))

Mainly, the last highlighted sentence, in conjunction with the others, creates the problem in this proof. It assumes the intersection between Y and Y' is non-empty, however, if $|X| = 2$ then the intersection between Y and Y' will indeed be the empty set. There is no logical reasoning for what is done when reaching the empty set.

If they tried to set their base case to $n = 0$, then this argument would simply fail at "Pick two distinct flowers $f, f' \in X$" when $|X| = 1$. Further proving that this argument is illogical, as it assumes impossible properties of sets.

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!

Base Case:

Let the base case be where $\langle b, \sigma \rangle \Downarrow \ False...$

$$\frac{D_1 :: \langle b, \sigma \rangle \Downarrow \ false}{\langle \texttt{while } b \texttt{ do } x := x + 2, \sigma \rangle \Downarrow \sigma}$$

Let c represent the command $(x := x + 2)$, and W represent the command `while` b `do` c

Pick arbitrary σ'' s.t. $D'' :: \langle W, \sigma \rangle \Downarrow \sigma''$.

By inversion and determinism of boolean expressions, D'' also uses the rule for `while false`, thus $\sigma'' = \sigma$

By definition, $\sigma(x)$ is even, thus $\sigma''(x)$ is even

Inductive Step:

We can now induct on the case where $\langle b, \sigma \rangle \Downarrow \ True...$

$$\frac{D_1 :: \langle b, \sigma \rangle \Downarrow \ true \qquad D_2 :: \langle x := x + 2, \sigma \rangle \Downarrow \ \sigma_1 \qquad D_3 :: \langle \texttt{while } b \texttt{ do } x := x + 2, \sigma_1 \rangle \Downarrow \sigma'}{\langle \texttt{while } b \texttt{ do } x := x + 2, \sigma \rangle \Downarrow \sigma'}$$

Let c represent the command $(x := x + 2)$, and W represent the command `while` b `do` c

Pick arbitrary σ'' s.t. $D'' :: \langle W, \sigma \rangle \Downarrow \sigma''$.

By inversion and determinism of boolean expressions, D'' also uses the rule for `while true`, and has subderivations $D_2'' :: \langle c, \sigma \rangle \Downarrow \sigma_1''$ and $D_3'' :: \langle W, \sigma_1'' \rangle \Downarrow \sigma''$

By inductive hypothesis on D_2 (with D_2''): $\sigma_1 = \sigma_1''$. We now have $D_3'' :: \langle W, \sigma_1 \rangle \Downarrow \sigma''$

By inductive hypothesis on D_3 (with D_3''): $\sigma' = \sigma''$

It suffices to show that $\sigma_1(x)$ is even, proving that arbitrary $\sigma_1''(x)$ is also even. If we show $\sigma_1(x)$ is even, then we have $D_3 :: \langle W, \sigma_1 \rangle \Downarrow \sigma'$ where $\sigma'(x)$ will be even by inductive reasoning provided the aforementioned base case and inductive step.

We have:

$\langle x := x + 2, \sigma \rangle \Downarrow \ \sigma_1$

$$\frac{\langle x + 2, \sigma \rangle \Downarrow n \qquad \sigma_1 = \sigma[x := n]}{\langle x := x + 2, \sigma \rangle \Downarrow \ \sigma_1}$$

Question assigned to the following page: 3

It now suffices to show $\langle x+2, \sigma\rangle \Downarrow n$ results in even n.

$$\frac{\langle x, \sigma\rangle \Downarrow \sigma(x) \qquad \langle 2, \sigma\rangle \Downarrow 2 \qquad n = \sigma(x) + 2}{\langle x+2, \sigma\rangle \Downarrow n}$$

By definition, $\sigma(x)$ is even. By definition of even numbers, 2 is even. By definition of integer addition, two even integers added together indeed stay even. Thus $\sigma(x) + 2 = n$ is indeed even.

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:

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \ \texttt{exc} \ n}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma' \ \texttt{exc} \ n} \ \textsf{seq1} \qquad \frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \quad \langle c_2, \sigma' \rangle \Downarrow t}{\langle c_1; c_2, \sigma \rangle \Downarrow t} \ \textsf{seq2}$$

We also introduce three additional commands:

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

- 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.

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

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

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n \quad \langle x := n, \sigma'\rangle \Downarrow \sigma'[x := n] \quad \langle c_2, \sigma'[x := n]\rangle \Downarrow \sigma''}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2,\ \sigma\rangle \Downarrow \sigma''}\ \mathsf{try}_{\mathsf{exception}}$$

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

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

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2,\ \sigma\rangle \Downarrow \sigma''\ \mathsf{exc}\ n}\ \mathsf{finally}_{\mathsf{exc}_1}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''\ \mathsf{exc}\ m}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2,\ \sigma\rangle \Downarrow \sigma''\ \mathsf{exc}\ m}\ \mathsf{finally}_{\mathsf{exc}_2}$$

Question assigned to the following page: 5

Exercise 2F-5. 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.

It is arguably far more elegant to describe "IMP with Exceptions" with small-step contextual semantics, primarily due to the pitfall large-step operational semantics encounter when faced with non-terminating command sequences. It is well-known that contextual semantics allows us to address the issue of potential non-terminating execution models. We can leverage this characteristic of contextual semantics by using it to describe "IMP with Exceptions", thus allowing us to identify exceptional command terminations, which may have otherwise gone unidentified in the scenario where we use large-step operational semantics. Through taking atomic steps through the execution of an "IMP with Exceptions" program, we are not only able to better reason about the behavior of our program during its execution, but we are also able to better incorporate the benefits of our newly introduced commands and exceptional termination states.

Perhaps one could make the counterargument that it "feels" more natural to describe "IMP with Exceptions", as the interpretability of the newly introduced commands and prior commands within IMP is far greater than compared to contextual semantics. Here I argue that it's not the interpretability of the rules themselves that matter the most, but rather it's the interpretability of an arbitrary execution model within our programming language which outweighs all else. The type of semantics used to define our programming language serves as a direct interface for this interpretation. Consequently, operational semantics is just far too constraining in the case of "IMP with Exceptions", for the reasons and by the arguments aforementioned.

No questions assigned to the following page.

Exercise 2C. Language Features, Coding. Download the Homework 2 code pack from the course web page. Modify `hw2.ml` so that it implements a complete interpreter for "IMP with exceptions (and `print`)". You may build on your code from Homework 1 (although the `let` command is not part of this assignment). Using OCaml's exception mechanism to implement IMP exceptions is actually slightly harder than doing it "naturally", so I recommend that you just implement the operational semantics rules. The `Makefile` includes a "`make test`" target that you should use (at least) to test your work.

Hint: to check if a termination `term` is an exception, use syntax like

```
begin match term with
| Normal -> do_something
| Exceptional(n) -> do_something_else using n
end
```

Modify the file `example-imp-command` so that it contains a "tricky" terminating IMP command (presumably involving exceptions) that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

Submission. Turn in the formal component of the assignment as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment via the `autograder.io` website.