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" `:-)`)

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!

Answer We will induct on the structure of the operational semantics of the while statement.

The base case corresponds to the case where the while loop terminates immediately. This happens when the condition b evaluates to false in the initial state σ. In this case, the while loop does not execute, and the state σ remains unchanged. Since σ is even at first and $\sigma = \sigma'$ in the base case, σ' is also even.

Question assigned to the following page: 3

For inductive step, there are two cases. One is that b is false, the other is b is true. When b is false, this case is the base case. We know the state won't be change when the while loop terminates. Since $\sigma(x)$ is even, $\sigma'(x)$ is even as well. When b is true, we execute $x := x + 2$. we know that if $\sigma(x)$ is even, then $\sigma'(x) = \sigma(x) + 2$, which is still even, because the sum of two even numbers is always even. Hence, $\sigma'(x)$ is even. Since the loop executes at least once, we are left with the same situation again:

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

will eventually terminate, and by the inductive hypothesis, the final value of $\sigma''(x)$ will also be even.

Thus, the invariant holds: each time the body of the loop executes, the value of x remains even. This ensures that $\sigma'(x)$ is even at the end of the loop execution.

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 $\textsf{throw}\ e$ command raises an exception with argument e.

Question assigned to the following page: 4

- 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.)

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.

Answer

$$\frac{\langle e, \sigma\rangle \Downarrow n}{\langle \mathsf{throw}\ e, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n}\ \mathsf{throw} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow t}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2, \sigma\rangle \Downarrow t}\ \mathsf{try\ normal}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n \quad \langle c_2, \sigma[x := n]\rangle \Downarrow t}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2, \sigma\rangle \Downarrow t}\ \mathsf{try\ exception} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow t}\ \mathsf{finally1}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n_1 \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n_1}\ \mathsf{finally2} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''\ \mathsf{exc}\ n_2}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow \sigma''\ \mathsf{exc}\ n_2}\ \mathsf{finally3}$$

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.

Answer Small-step semantics naturally captures the idea of how each individual statement or expression in a program progresses from one state to another. When dealing with exceptions, this step-by-step approach is very useful, because exceptions are often raised or handled in the middle of executing a program, and the program state transitions can be very fine-grained. Small-step semantics can model the moment when an exception is raised (as a state transition) and when it is caught or handled.

In IMP with exceptions, an exception can be thrown at any point during the execution of a program. Small-step semantics allows for a more explicit, clear representation of what happens at each point in time. For example, a small-step rule could be defined to handle the transition of an exception being thrown and the effect this has on the program's control flow, whereas a large-step (big-step) semantics might attempt to abstract this transition away, making it less clear when exactly the exception is thrown or how the program responds to it.

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.