Exercise 1: Find the flaw in the following proof by induction of the fact that “All flowers smell the same” (that is, if you think this proof is flawed!). Please indicate exactly which sentences are wrong in my proof. Bringing me a counterexample does not constitute an acceptable solution.

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.

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

(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:** $n = 1$. Obviously all singleton sets of flowers contain flowers that smell the same.

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

Exercise 2: Prove by induction the following statement about the operational semantics:

For any boolean command $b$ and any initial state $\sigma$ such that $\sigma(x)$ is even, if $\langle \text{while } b \text{ do } x := x + 2, \sigma \rangle \Downarrow \sigma'$ then $\sigma'(x)$ is even. Make sure you state what you induct on, what is the base case and what are the inductive cases. Show representative cases among the latter. Do not do a proof by mathematical induction!
Exercise 3: Language Feature: Local Variables

Consider the IMP language with a new command construct

\[
\text{let } x = e \text{ in } c
\]

The informal semantics of this construct is that the Aexp \(e\) is evaluated and then a new local variable \(x\) is created with lexical scope \(c\) and initialized with the result of evaluating \(e\). Then the command \(c\) is evaluated. We also extend IMP with a new command “\text{print } e” which evaluates the Aexp \(e\) and “displays the result” in some un-modeled manner but is otherwise similar to \text{skip}.

We expect (the curly braces are syntactic sugar):

\[
x := 1 ;
y := 2 ;
\{ \text{let } x = 3 \text{ in}
    \text{print } x ;
    \text{print } y ;
    x := 4 ;
y := 5
\} ;
\text{print } x ;
\text{print } y
\]

to display “3 2 1 5”.

- Extend the natural-style operational semantics judgment \(\langle c, \sigma \rangle \Downarrow \sigma'\) with one new rule for dealing with the \text{let} command. Pay careful attention to the scope of the newly declared variable and to changes to other variables.
- Extend the set of redexes, contexts and reduction rules for the contextual-style operational semantics that we discussed in class to account for the \text{let} command.

Exercise 4: Language Feature: Exceptions

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 \(n \in \mathbb{Z}\)):

\[
T ::= \sigma \quad \text{normal termination} \\
| \sigma \text{ exc } n \quad \text{exceptional termination}
\]

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

\[
\langle c, \sigma \rangle \Downarrow \sigma' \text{ exc } n
\]
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 $\sigma'$. 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:

$$
\frac{\langle c_1, \sigma \rangle \downarrow \sigma' \text{ exc } n}{\langle c_1; c_2, \sigma \rangle \downarrow \sigma' \text{ exc } n} \quad \frac{\langle c_1, \sigma \rangle \downarrow \sigma' \quad \langle c_2, \sigma' \rangle \downarrow t}{\langle c_1; c_2, \sigma \rangle \downarrow t}\quad \text{seq1 seq2}
$$

We also introduce three additional commands:

1. **throw** $e$: raises an exception with argument $e$.
2. **try** $c_1$ **catch** $x$ **catch** $c_2$: 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.
3. **after** $c_1$ **finally** $c_2$: 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. We thus expect:

```plaintext
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 big-step operational semantics inference rules using the new judgment for the three new commands presented above. You should present six (6) new rules total.

• Download the Homework 1 code pack from the course web page hw1.tgz. Modify hw1.ml so that it implements a complete interpreter for “IMP with exceptions (and print)”. You need not implement the let command.

  Do not use OCaml’s exception mechanism to implement IMP exceptions (it is actually easier to directly implement the operational semantics rules. The Makefile includes a “make test” target that you should use to test your work.

• Modify the file example-imp-command so that it contains a “tricky” IMP command involving exceptions that can be parsed by our IMP test harness. For example, the following shell command should not yield a parse error.

  imp < example-imp-command

• Rename hw1.ml to firstname_lastname_hw1.ml and rename example-imp-command to firstname_lastname_example-imp-command and email them to me. Do not modify any other files. Your submission’s grade will be based on how many of the submitted example-imp-commands it interprets correctly (in a manner just like the “make test” trials). Extra Credit If your submitted example-imp-command breaks the greatest (non-zero) number of interpreters.