Exercise 1: Bookkeeping. Indicate in a sentence or two how much time you spent on this homework, how difficult you found it subjectively, and what you found to be the hardest part. Any non-empty answer will receive full credit.

If there is something else you would like to share about yourself, please do so. Also, if your opinions have changed since the last homework, indicate one thing you like about the class so far and one thing you would change about it.

Exercise 2: Mathematical Induction. Find the flaw in the following inductive proof that “All flowers smell the same”. Please indicate exactly which sentences are wrong in the proof. Bringing me a counterexample does not constitute an acceptable solution.

Proof: Let $F$ be the set of all flowers, and let $\text{smells}(f)$ be the smell of the flower $f \in F$ (the range of $\text{smells}$ is not so important, but we will assume that it admits equality). We will 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.

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

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 will 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
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 ∈ Y ∩ Y'. Obviously, x ≠ f and x ≠ 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.

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

Exercise 3: Induction. Prove by induction the following statement about the big-step operational semantics of IMP:

For any BExp b and any initial state σ such that σ(x) is even, if

<while b do x := x + 2, σ> → σ'

then σ'(x) is even.

Make sure you state what you induct on and where you apply the induction hypothesis. Do not do a proof by mathematical induction! (Use some other kind of induction.)

Exercise 4: Induction. Recal the following machine defined in the lecture for executing integer arithmetic instructions.

<push(n) :: is, s, ns> → <is, s, n :: ns>
<load(x) :: is, s, ns> → <is, s, s(x) :: ns>
<add :: is, s, n2 :: n1 :: ns> → <is, s, (n1 + n2) :: ns>

The zero-or-more step relation is then defined as follows.

<is, s, ns> →* <is', s, ns'><is', s, ns'> →* <is'', s, ns''>
<is, s, ns> →* <is', s, ns'>
<is', s, ns'> →* <is'', s, ns''>

Prove by induction the following lemma that was used in the proof of correctness of the compiler during the lecture.

If <is1, s, []> →* <[], s, ns1> and <is2, s, ns1> →* <[], s, ns2>,
then <is1@is2, s, []> →* <[], s, ns2>.
Exercise 5: Language Features. We consider again IMP extended with a notion of integer-valued exceptions (or run-time errors).

In this exercise, we extend the structural small-step operational semantics for exceptions. To do so, we do not change the judgment forms. Thus, we define one-step evaluation judgments of the following forms:

\[
\begin{align*}
  e & \rightarrow_{\sigma} e' & \text{arithmetic expression } e \text{ steps to } e' \text{ in state } \sigma \\
  b & \rightarrow_{\sigma} b' & \text{boolean expression } b \text{ steps to } b' \text{ in state } \sigma \\
  \langle c, \sigma \rangle & \rightarrow \langle c', \sigma' \rangle & \text{command } c \text{ in state } \sigma \text{ steps to } c' \text{ in } \sigma'
\end{align*}
\]

However, we do add another terminal command (i.e., “command value”): \texttt{throw n}, which indicates an uncaught exception with value \( n \). Stated judgmentally, we write the judgment

\[ c \text{ terminal} \]

to mean that command \( c \) is a terminal command. This judgment is then defined with the following two inference rules:

\[
\begin{array}{c}
\text{skip terminal} \\
\text{throw n terminal}
\end{array}
\]

Note that our previous evaluation rules for commands must be extended to account for exceptions. For example, we have the following rules for sequencing:

\[
\begin{align*}
  \langle c_1, \sigma \rangle & \rightarrow \langle c_1', \sigma' \rangle \\
  \langle c_1; c_2, \sigma \rangle & \rightarrow \langle c_1'; c_2, \sigma' \rangle \\
  \langle \text{skip}; c_2, \sigma \rangle & \rightarrow \langle c_2, \sigma \rangle \\
  \langle \text{throw n}; c_2, \sigma \rangle & \rightarrow \langle \text{throw n}, \sigma \rangle
\end{align*}
\]

where the top two rules are as before, while the bottom rule is needed for exceptions. This bottom rule says that if the first command is a thrown exception, then the second command does not execute and the exception is propagated.

We introduce three additional commands for exceptions:

\[
\begin{align*}
  c & ::= \ldots \\
  & \quad | \text{throw n} \\
  & \quad | \text{try } c_1 \text{ catch } x \ c_2 \\
  & \quad | \text{after } c_1 \text{ finally } c_2
\end{align*}
\]

• The \texttt{throw n} command raises an exception with argument \( n \). For simplicity, we fix the argument to be an integer (rather than an arbitrary arithmetic expression as in Homework 2).
• 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 $n$, the variable $x \in L$ is assigned the value $n$ 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 $n_1$, then $c_2$ is executed:
  
  - If $c_2$ terminates normally, the `finally` command terminates by throwing an exception with value $n_1$ (i.e., the original exception $n_1$ is re-thrown at the end of the `finally` block, as in Java).
  - If $c_2$ throws an exception with value $n_2$, the `finally` command terminates by throwing an exception with value $n_2$ (i.e., the new exception $n_2$ overrides the original exception $n_1$, also as in Java).

These constructs are intended to have the standard exception semantics from languages like Java, C#, or ML, 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`.

• Give the structural small-step operational semantics inference rules for the three new exception commands.

• Starting with a structural small-step operational semantics for IMP as in Homework 2, do we need any additional rules besides the ones given in the previous part and the new sequencing rule given above to describe the semantics of exceptions fully? If so, give the additional rules. If not, briefly explain why not.

• Download the Homework 3 code pack from the course web page. The `README.txt` describes the code pack, like in Homework 2. Modify `hw3.ml` so that it implements a complete interpreter for “IMP with exceptions (and `print`)” following a small-step operational semantics. The `Makefile` includes a “make test” target that you should use (at least) to test your work.

To simplify this exercise, you do not need to implement `finally`, and it has been removed from the code pack. Note that you are asked to give inference rules for `finally` in the first part of this exercise.

The driver has a flag `--trace` that will print a trace of evaluation using your `step.com` function. For example, running “`imp --trace < example.imp`” will interpret the code and print a trace of each step of evaluation. Your interpreter will be tested by comparing traces.
• Modify the file example.imp so that it contains a “tricky” IMP command (presumably involving exceptions) that can be parsed by our IMP test harness (e.g., “imp < example.imp” should not yield a parse error).

• Rename hw3.ml to your_identkey-hw3.ml and rename example.imp to your_identkey-example.imp for submission. Do not modify any other files. Your submission’s grade will be based on how many of the submitted example.imps it interprets correctly (in a manner just like the “make test” trials). If your submitted example.imp breaks the greatest number of interpreters (and more than 0!), you will receive extra credit. If there is a tie, all those in the tie will receive the extra credit.

• Make sure your code compiles. Code that does not compile will not be graded. If there is some case you cannot get to work, simply comment it out.