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. Also, let’s get to know each other better, tell me something about yourself that I do not already know. Any non-empty answer will receive full credit.

Additionally, 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: Language Feature Design. Recall the extension to the IMP language with the let construct:

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

The informal semantics of let is that \( e \) is evaluated; then a new local variable \( x \) is created with lexical scope \( c \) and initialized with the result of evaluating \( e \); finally, the command \( c \) is evaluated. Recall the example from before, we expect

\[
\begin{align*}
x & := 1 ; \\
y & := 2 ; \\
\{ \text{let } x = 3 \text{ in} \\
& \quad \text{print } x ; \\
& \quad \text{print } y ; \\
& \quad x := 4 ; \\
& \quad y := 5 \\
\} ; \\
\text{print } x ; \\
\text{print } y
\end{align*}
\]

to display “3 2 1 5”.

For this exercise,

- Extend the set of redexes \( r \), contexts \( H \), and reduction rules for the contextual-style small-step operational semantics that we discussed in class to account for the `let` command.

- (Extra Credit). In the small-step operational semantics for IMP that we discussed in class, we needed to use the notion of state. With local declarations, the state can be kept as part of the program: the current values of the variables can be part of the local declarations for those variables. For example, we can encode the current command \( c \) along with the state \([x = 5, y = 6]\) as follows:

\[
\text{let } x = 5 \text{ in let } y = 6 \text{ in } c
\]

This has the advantage that both the state transformations and the program transformations are presented in a uniform way as program transformations. Show a modified form of contextual small-step semantics using this strategy for encoding the state as part of the program (i.e., define the redexes \( r \), the contexts \( H \), and the reduction rules).

Since we have eliminated the state, the new judgment forms for one-step evaluation are as follows:

- \( e \rightarrow e' \) arithmetic expression \( e \) steps to \( e' \)
- \( b \rightarrow b' \) boolean expression \( b \) steps to \( b' \)
- \( c \rightarrow c' \) command \( c \) steps to \( c' \)

And analogously, the local reduction judgment for commands has the following form:

\[
\langle r \rangle \rightarrow \langle c', \sigma' \rangle \quad \text{redex \( r \) reduces to } c'
\]

You only need to consider commands (i.e., you may assume the redexes, contexts, and local reduction rules for arithmetic and boolean expressions are given).

**Exercise 3: Structural Small-Step Operational Semantics.** In class, we defined a small-step operational semantics for IMP in the contextual-style. In this exercise, we define a small-step operational semantics for IMP using the structural-style. To do so, we consider the following three judgment forms for arithmetic expressions \( e \), boolean expressions \( b \), and commands \( c \):

- \( e \rightarrow_{\sigma} e' \) arithmetic expression \( e \) steps to \( e' \) in state \( \sigma \)
- \( b \rightarrow_{\sigma} b' \) boolean expression \( b \) steps to \( b' \) in state \( \sigma \)
- \( \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle \) command \( c \) in state \( \sigma \) steps to \( c' \) in \( \sigma' \)
Intuitively, the above three judgments capture one-step of evaluation for arithmetic expressions, boolean expressions, and commands, respectively. They are analogous to the small-step evaluation judgments used in class.\textsuperscript{1}

Let us define the one-step evaluation judgment of arithmetic expressions. We have a rule for looking up variables:

\[
\frac{}{x \rightarrow_{\sigma} \sigma(x)}
\]

There are no rules for constants \(n\), as they are values. For \(e_1 + e_2\), we need an inference rule corresponding to local reduction (called \textit{instructions} in Harper):

\[
n_1 + n_2 \rightarrow_{\sigma} n_1 + n_2
\]

Note that + on the left is the syntactic plus, while + on the right corresponds to the addition of two integers. To get to the reduction step in the rule above, we use the following rules (called \textit{search rules} in Harper):

\[
\begin{align*}
e_1 & \rightarrow_{\sigma} e_1' \\
e_2 & \rightarrow_{\sigma} e_2' \\
e_1 + e_2 & \rightarrow_{\sigma} e_1' + e_2 \\
n_1 + e_2 & \rightarrow_{\sigma} n_1 + e_2'
\end{align*}
\]

Observe that these rules specify a left-to-right evaluation of \(e_1 + e_2\). The rules for \(e_1 - e_2\) and \(e_1 * e_2\) are similar.

For this exercise,

- Define the one-step evaluation judgment for IMP commands

\[
\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle
\]

(i.e., give inference rules) in the structural-style that is analogous to the small-step semantics in the contextual-style discussed in class. You only need to consider the core IMP language of commands (i.e., you do not need to consider \texttt{let}). You may assume that the one-step evaluation judgments for arithmetic and boolean expressions have been defined (i.e., \(e \rightarrow_{\sigma'} e'\) and \(b \rightarrow_{\sigma'} b'\) are given).

- Discuss how your structural small-step operational semantics for IMP commands corresponds to the contextual small-step operational semantics given in class.

\textsuperscript{1}Here, we have changed the judgment forms for arithmetic expressions and boolean expressions slightly to highlight that the state is constant throughout expression evaluation. In Winskel, the analogous judgments are written as follows:

\[
\begin{align*}
\langle e, \sigma \rangle \rightarrow_1 \langle e', \sigma' \rangle \\
\langle b, \sigma \rangle \rightarrow_1 \langle b', \sigma' \rangle \\
\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle
\end{align*}
\]
1. What do the search rules correspond to in the contextual-style?
2. Briefly explain why they should behave equivalently. We do not yet have the tools to prove this statement, so an informal discussion is fine.

**Exercise 4: More Language Feature Design.** We now extend IMP with a notion of integer-valued exceptions (or run-time errors), as in Java, C#, or ML. 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 then redefine our big-step operational semantics judgment as follows:

\[
\langle c, \sigma \rangle \Downarrow T
\]

where 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”) that is set when the exception is thrown and available when the exception is caught.

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

\[
\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \text{ exc } n}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma' \text{ exc } n} \quad \text{seq}_{\text{exc}}
\]

\[
\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \quad \langle c_2, \sigma' \rangle \Downarrow T}{\langle c_1; c_2, \sigma \rangle \Downarrow T} \quad \text{seq}_{\text{norm}}
\]

We also introduce three additional commands:

\[
c ::= \ldots \\
| \text{throw } e \\
| \text{try } c_1 \text{ catch } x \ c_2 \\
| \text{after } c_1 \text{ finally } c_2
\]

- 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 \( n_1 \), the variable \( x \in L \) is **assigned** the value \( n_1 \) 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 value \( n_2 \) overrides the original exception value \( n_1 \), 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. We thus expect the following:

\[
\begin{align*}
x &:= 0 \\
\{ \text{try} \\
& \quad \text{if } x \leq 5 \text{ then throw 33 else throw 55} \\
& \quad \text{catch } x \\
& \quad \quad \text{print } x \\
\} \\
\text{while true do} \\
& \quad x := x - 15 \\
& \quad \text{print } x \\
& \quad \text{if } x \leq 0 \text{ then throw (x*2) else skip}
\end{align*}
\]

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 our new judgment) for the three new commands presented here. You should present six (6) new rules total.

• Argue for or against the claim that it would be more natural to describe “IMP with exceptions” using small-step contextual operational 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.

• Download the Homework 2 code pack from the course web page. The README.txt describes the code pack, like in Homework 1. 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 big-step operational semantic 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

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

- 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 hw2.ml to your_identikey-hw2.ml and rename example.imp to your_identikey-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.