The purpose of this assignment is to make a connection between big-step and small-step operational semantics. We will focus only structural operational semantics in this homework. Recall the evaluation guideline from the course syllabus.

Both your ideas and also the clarity with which they are expressed matter—both in your English prose and your code!

We will consider the following criteria in our grading:

• How well does your submission answer the questions? For example, a common mistake is to give an example when a question asks for an explanation. An example may be useful in your explanation, but it should not take the place of the explanation.

• How clear is your submission? If we cannot understand what you are trying to say, then we cannot give you points for it. Try reading your answer aloud to yourself or a friend; this technique is often a great way to identify holes in your reasoning. For code, not every program that "works" deserves full credit. We must be able to read and understand your intent. Make sure you state any preconditions or invariants for your functions.

Submission Instructions. Upload to the moodle exactly three files named as follows:

• hw2-YourIdentiKey.pdf with your answers to the written questions. Scanned, clearly legible handwritten write-ups are acceptable. Please no other formats—no .doc or .docx. You may use whatever tool you wish (e.g., \LaTeX, Word, markdown, plain text, pencil+paper) as long as it is legibly converted into a pdf.

• hw2-YourIdentiKey.ml with your OCaml implementation.

• hw2-YourIdentiKey.imp with your IMP test case.

1. Feedback. Complete the survey on the linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.

2. Short-Circuit Evaluation. Consider the Boolean expressions:

\[ b_1 \&\& b_2 \] and \[ b_1 \| b_2 \]

for logical-and and logical-or in IMP.
(a) Define small-step reduction rules for the logical-and and logical-or expressions. Give rules such that evaluation of both the expressions proceeds left-to-right and short-circuits.

Note that the left-to-right, short-circuit, big-step evaluation rules for logical-and are given in Winskel Chapter 2. The left-to-right, short-circuit, big-step evaluation rules for logical-or is Exercise 2.3.

Hint: The OCaml implementation in the code pack implements left-to-right, short-circuit evaluation in both the big-step and the small-step interpreters.

(b) For logical-and in IMP, does short-circuiting versus non-short-circuiting affect the derivability in the big-step semantics? In other words, considering the set of rules defining the big-step semantics using short-circuiting logical-and versus the set of rules defining the big-step semantics using non-short-circuiting logical-and, is there a judgment that is derivable in one system but not the other? If yes, provide such a judgment and brief explanation. If not, give a brief explanation why there is no difference in derivability.

(c) How about for small-step semantics? Please be clear and concise.

3. Local Variables. Recall the extension to IMP commands with

\[ c ::= \cdots | \text{let } x = a \text{ in } c \]

Recall that the informal semantics of this construct is that the \( a \in \text{Aexp} \) is evaluated and then a new local variable \( x \) is created with lexical scope \( c \) and initialized with the result of evaluating \( a \). Then the command \( c \) is evaluated.

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”. The curly braces are syntactic sugar to “parenthesize” commands.

- Extend the structural small-step operational semantics judgment form \( \langle c, \sigma \rangle \rightarrow \langle c, \sigma' \rangle \) with new rules for dealing with the let command. Pay careful attention to the scope of the newly declared variable and to changes to other variables—really, this is tricky!

Hint: You are allowed to allocate “fresh” locations. You can state that a location is fresh with respect to a state as follows:

\[ x \notin \text{dom}(\sigma), \]

which states that location \( x \) is not in domain of state \( \sigma \).
4. **Exceptions.** We now extend IMP with a notion of integer-valued exceptions (or run-time errors), as in Java, C#, or ML.

   (a) **Big-Step Operational Semantics.** We introduce a new type $T$ to represent command terminations, which can either be normal or exceptional (with an exception value $n \in \mathbb{Z}$):

   $$ T ::= \sigma \quad \text{normal termination} \\
   \quad \mid \sigma \text{exc } n \quad \text{exceptional termination} $$

   We then redefine our big-step operational semantics judgment form 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 now have the following rules for sequencing:

   $$ \begin{array}{c}
   \frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \text{exc } n}{\langle c_1;c_2, \sigma \rangle \Downarrow \sigma' \text{exc } n} &\quad \frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \quad \langle c_2, \sigma' \rangle \Downarrow T}{\langle c_1;c_2, \sigma \rangle \Downarrow T}
   \end{array} $$

   We also introduce three additional commands:

   $$ c ::= \ldots \\
   \mid \text{throw } e \\
   \mid \text{try } c_1 \text{ catch } x \ c_2 \\
   \mid \text{after } c_1 \text{ finally } c_2 $$

   • 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 $x$ 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:

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

to output “33 18 3 -12” and then terminate with an uncaught exception with value -24.

i. 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.

ii. Does the big-step operational semantic inference rule for the 

\textbf{let } x = a \textbf{ in } c

command requires revising in the presence of exceptions? If so, give new rule(s) and briefly explain the changes that are needed. If not, briefly explain why not. Note that the scope of \( x \) should be \( c \) regardless of whether or not an exception is thrown in \( c \).

(b) \textbf{Small-Step Operational Semantics.} We now extend the structural small-step operational semantics for exceptions. To do so, we do not need to change the judgment forms. Thus, we define one-step evaluation judgments of the following forms:

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

However, we do add another terminal command (i.e., “command value”): 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:

\[
\begin{array}{c}
  \text{skip terminal} \\
  \text{throw } n \text{ terminal}
\end{array}
\]
Note that our previous evaluation rules for commands must be extended to account for exceptions. For example, we have the following rules for sequencing:

\[
\begin{align*}
\langle c_1; c_2, \sigma \rangle &\rightarrow \langle c'_1; c_2, \sigma' \rangle \\
\langle c_1, \sigma \rangle &\rightarrow \langle c'_1, \sigma' \rangle \\
\langle \text{skip}; c_2, \sigma \rangle &\rightarrow \langle c_2, \sigma \rangle \\
\langle \text{throw } n; c_2, \sigma \rangle &\rightarrow \langle \text{throw } n, \sigma \rangle
\end{align*}
\]

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.

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

ii. Do the small-step operational semantic inference rules for the \textit{let} \( x = a \) \textit{in} \( c \) command from Question 3 require revising in the presence of exceptions? If so, give new rule(s) and briefly explain the changes that are needed. If not, briefly explain why not. Note that the scope of \( x \) should be \( c \) regardless of whether or not an exception is thrown in \( c \).

iii. Argue for or against the claim that it would be more natural to describe “IMP with exceptions” using small-step operational semantics (versus big-step). 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.

(c) Language Implementation. In this exercise, you will implement an interpreter for IMP extended with \textit{let} and exceptions.

i. 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 complete interpreters for “IMP with \textit{let} and exceptions” (one based on big-step and one based on small-step). You may build on your code from Homework 1 as a starting point if you wish.

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/small-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

\[
\text{begin match term with} \\
| \text{Normal sigma} -> (* do_something *) \\
| \text{Exceptional (sigma, n)} -> (* do_something_else *) \\
\text{end}
\]
ii. 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).

Do not modify any other files. Your submission’s grade will be based on how many of the submitted example.imp’s 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.

The driver has flags --no-big and --small to select whether or not your big-step or small-step interpreters are executed. It also has a flag --trace that will print a trace of evaluation using your step_com function (if --small is enabled). For example, running “./imp --small --no-big --trace < example.imp” will interpret the code and print a trace of each step of evaluation. Your interpreter will be tested by comparing traces. Use “./imp --help” for more information on the flags.

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

   For any $b \in \text{Bexp}$ and any initial state $\sigma$ such that $\sigma(x)$ is even, if
   
   $$\langle \textbf{while} \ b \ \textbf{do} \ x := x + 2, \sigma \rangle \downarrow \sigma'$$

   then $\sigma'(x)$ 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.