The purpose of this assignment is to introduce non-determinism in operational semantics and to gain further experience with induction on derivations.

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:

• hw4-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.

• hw4-YourIdentiKey.ml with your OCaml implementation.

• hw4-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. Induction. Prove by induction the following statement about the denotational semantics of IMP.
For any $b \in \text{Bexp}$ and any initial state $\sigma$ such that $\sigma(x)$ is even, if

$$[\text{while } b \text{ do } x := x + 2] \sigma = \sigma',$$

then $\sigma'(x)$ is even.

Note that we are assuming that $[\text{while } b \text{ do } x := x + 2] \sigma \neq \bot$ in the above. Unlike in the previous assignment, this time you should use denotational semantics for the proof.

If you wish, you may consider the simplified language as in class in which case the problem can be rephrased as follows:

For any $b \in \text{Bexp}$ and any initial state $n$ such that $n$ is even, if

$$[\text{while } b \text{ do } x := x + 2] n = n',$$

then $n'$ is even.

Note that we are assuming that $[\text{while } b \text{ do } x := x + 2] n \neq \bot$ in the above.

**Hint:** your proof should proceed by mathematical induction.

3. **Operational and Denotational Semantics.** Regular expressions are commonly used as abstractions for string matching. Here is an abstract syntax for regular expressions:

$$r ::= \text{`}c\text{'} \quad \text{singleton – matches the character } c$$

$$| \text{empty} \quad \text{skip – matches the empty string}$$

$$| r_1 r_2 \quad \text{concatenation – matches } r_1 \text{ followed by } r_2$$

$$| r_1 | r_2 \quad \text{or – matches } r_1 \text{ or } r_2$$

$$| r^* \quad \text{Kleene star – matches 0 or more occurrences of } r$$

$$| \text{.} \quad \text{matches any single character}$$

$$| [\text{`}c_1\text{'} - \text{`}c_2\text{'}] \quad \text{matches any character between } c_1 \text{ and } c_2 \text{ inclusive}$$

$$| r^+ \quad \text{matches 1 or more occurrences of } r$$

$$| r? \quad \text{matches 0 or 1 occurrence of } r$$

We will call the first five cases the *primary* forms of regular expressions. The last four cases can be defined in terms of the first five. We also give an abstract grammar for strings (modeled as lists of characters):

$$s ::= \text{nil} \quad \text{empty string}$$

$$| c :: s \quad \text{string with first character } c \text{ and other characters } s$$

We write “bye” as shorthand for $\text{`}b\text{'} :: \text{`}y\text{'} :: \text{`}e\text{'} :: \text{nil}.$

We introduce the following big-step operational semantics judgment for regular expression matching:

$$\Downarrow r \text{ matches } s \text{ leaving } s'$$
The interpretation of the judgment is that the regular expression \( r \) matches some prefix of the string \( s \), leaving the suffix \( s' \) unmatched. If \( s' = \text{nil} \), then \( r \) matched \( s \) exactly. For example,

\[ \vdash 'h'('e'+) \text{ matches } "hello" \text{ leaving } "llo" \]

Note that this operational semantics may be considered \textit{non-deterministic} because we expect to be able to derive all three of the following:

\[ \vdash ('h'|'e')* \text{ matches } "hello" \text{ leaving } "ello" \]
\[ \vdash ('h'|'e')* \text{ matches } "hello" \text{ leaving } "hello" \]
\[ \vdash ('h'|'e')* \text{ matches } "hello" \text{ leaving } "llo" \]

We leave the rules of inference defining this judgment unspecified. You may consider giving this set of inference rules an optional exercise.

Instead, we will use \textit{denotational semantics} to model the fact that a regular expression can match a string leaving many possible suffixes. Let \( S \) be the set of all strings, let \( \wp(S) \) be the powerset of \( S \), and let \( \text{RE} \) range over regular expressions. We introduce a semantic function:

\[ \mathcal{R}[\cdot] : \text{RE} \rightarrow (S \rightarrow \wp(S)) \]

The interpretation is that \( \mathcal{R}[r] \) is a function that takes in a string-to-be-matched and returns a set of suffixes. We might intuitively define \( \mathcal{R} \) as follows:

\[ \mathcal{R}[r](s) = \{ s' | \vdash r \text{ matches } s \text{ leaving } s' \} \]

In general, however, one should not define the denotational semantics in terms of the operational semantics. Here are two correct semantic functions:

\[ \mathcal{R}[c'](s) = \{ s' | s = 'c' :: s' \} \]
\[ \mathcal{R}[	ext{empty}](s) = \{ s \} \]

(a) Give the denotational semantics functions for the other three primal regular expressions. Your semantics functions \textit{may not} reference the operational semantics.

(b) We want to update our operational semantics for regular expressions to capture multiple suffixes. We want our new operational semantics to be deterministic—it should give the same answer as the denotational semantics above. We introduce a new judgment as follows:

\[ \vdash r \text{ matches } s \text{ leaving } S \]

And use rules of inference like the following:

\[ \vdash 'c' \text{ matches } s \text{ leaving } \{ s' | s = 'c' :: s' \} \]
\[ \vdash \text{empty matches } s \text{ leaving } \{ s \} \]
\[ \vdash r_1 \text{ matches } s \text{ leaving } S_1 \quad \vdash r_2 \text{ matches } s \text{ leaving } S_2 \]
\[ \vdash r_1 | r_2 \text{ matches } s \text{ leaving } S_1 \cup S_2 \]

Do one of the following:
• Either give operational semantics rules of inference for \( r^* \) and \( r_1 \ r_2 \). Your operational semantics rules may not reference the denotational semantics. You may not place a derivation inside a set constructor, as in: \( \{ x \mid \exists y. \vdash r \text{ matches } x \text{ leaving } y \} \). Each inference rule must have a finite and fixed set of hypotheses.

• Or argue in one or two sentences that it cannot be done correctly in the given framework. Back up your argument by presenting two attempted but “wrong” rules of inference and show that each one is either unsound or incomplete with respect to our intuitive notion of regular expression matching.

Part of doing research in any area is getting stuck. When you get stuck, you must be able to recognize whether “you are just missing something” or “the problem is actually impossible.”

(c) Download the Homework 4 code pack from the course web page. The README.txt describes the code pack. Write an interpreter for regular expressions based on the denotational semantics. In particular, you must write a function

\[
\text{matches}: \text{re} \rightarrow \text{re_string} \rightarrow \text{stringset}
\]

in \text{hw4.ml}, which corresponds exactly to your semantic function

\[
\mathcal{R}[\cdot] : \text{RE} \rightarrow S \rightarrow \wp(S).
\]

Your interpreter should handle all of the regular expression forms, not just the primal ones. The simplest method for implementing the non-primal forms are to define them in terms of the primal ones. The Makefile includes a “make test” target that you should use (at least) to test your work. Modify the example.re file to include your best test case. Do not modify any other files.

4. **Final Project.** Refresh your memory about the final project from the course website and form a group. You may use Piazza to discuss project ideas and find a group. From my perspective, the ideal group size is two, but I will consider larger groups on a case-by-case basis.

On Piazza, for each group, post a short paragraph on some initial thoughts for a project. We will look for a more detailed project proposal in the weeks to come.