Exercise 3F-1. Regular Expression, Large-Step [10 points].

Large step operational semantics rules of inference for the remaining three forms of regular expressions are as below:

1. **Concatenation:**
   (e1 e2)

   Here, to match a concatenation of e1 and e2, we must match e1 with a prefix of s, leaving a suffix s'. Then, e2 must match with s', leaving the final unmatched suffix s''.

$$\frac{\text{`}e_1 \text{ matches } s \text{ leaving } s' \quad \text{`}e_2 \text{ matches } s' \text{ leaving } s''}{\text{`}e_1 e_2 \text{ matches } s \text{ leaving } s''}$$

2. **Or (e1 | e2)**

   For this, to match e1 | e2, the input string s can either match the e1 or match the e2. Thus either of the below paths can be chosen (this becomes non-deterministic)

$$\frac{\text{`}e_1 \text{ matches } s \text{ leaving } s'}{\text{`}e_1 | e_2 \text{ matches } s \text{ leaving } s'}$$

   Or

$$\frac{\text{`}e_2 \text{ matches } s \text{ leaving } s'}{\text{`}e_1 | e_2 \text{ matches } s \text{ leaving } s'}$$

3. **Kleene Star (e*)**

   To match e*, we can either match zero occurrences (leaving s as is) or match one occurrence of e, followed by e* on the remainder:

   So we have the following ways,

   - **Zero Occurrences: (Base Case)**

$$\frac{}{\text{`}e * \text{ matches } s \text{ leaving } s}$$

   - **One or more occurrences:**

$$\frac{\text{`}e \text{ matches } s \text{ leaving } s' \quad \text{`}e * \text{ matches } s' \text{ leaving } s''}{\text{`}e * \text{ matches } s \text{ leaving } s''}$$

For example, Matching (”h” | ”e”)∗ with "hello"

Peer Review ID: 311012197 — enter this when you fill out your peer evaluation via gradescope

Regular Expression: (”h” | ”e”)∗

Input String: "hello" ("h" :: "e" :: "l" :: "l" :: "o" :: nil)

Deriving as follows,

1. Using the Kleene Star (e*) Rule:

$$\frac{}{`(\text{"}h\text{"}|\text{"}e\text{"})*\ \text{matches}\ \text{"}hello\text{"}\ \text{leaving}\ \text{"}hello\text{"}}$$

   Here, *Zero occurrences match directly, leaving the entire input string unmatched.*

2. Matching One or More Occurrences:

$$\frac{`(\text{"}h\text{"}|\text{"}e\text{"})\ \text{matches}\ \text{"}hello\text{"}\ \text{leaving}\ \text{"}ello\text{"} \quad `(\text{"}h\text{"}|\text{"}e\text{"})*\ \text{matches}\ \text{"}ello\text{"}\ \text{leaving}\ \text{"}ello\text{"}}{`(\text{"}h\text{"}|\text{"}e\text{"})*\ \text{matches}\ \text{"}hello\text{"}\ \text{leaving}\ \text{"}ello\text{"}}$$

   Here, we match the first “h” and leave back the “ello”

3. Applying the Or (e1 | e2) Rule:

$$\frac{`\text{"}h\text{"}\ \text{matches}\ \text{"}hello\text{"}\ \text{leaving}\ \text{"}ello\text{"}}{`(\text{"}h\text{"}|\text{"}e\text{"})\ \text{matches}\ \text{"}hello\text{"}\ \text{leaving}\ \text{"}ello\text{"}}$$

   The | here allows us to choose “h” as the match thereby satisfying the rule

Exercise 3F-2. Regular Expression and Sets [5 points].

So, here I am trying to attempt to develop a deterministic operational semantics rules for e1 e2 (concatenation) and e*(Kleene star) and see if it is possible or impossible.

1. **Concatenation (e1 e2)**

   When matching e1e2, the intermediate suffixes generated by e1 need to be fed into e2.
   Thus,
   Attempted Rule for concatenation:

$$\frac{`e_1\ \text{matches}\ s\ \text{leaving}\ S, \quad \forall s' \in S, `e_2\ \text{matches}\ s'\ \text{leaving}\ S'}{`e_1e_2\ \text{matches}\ s\ \text{leaving}\ \bigcup_{s'\in S} S'}$$

   Here, we can clearly see that the rule is invalid. The rule essentially requires evaluating e2 for every suffix s" derived from e1, which means we cannot perform this operation with a finite and fixed set of hypotheses. It leads to a scenario where the rule cannot capture the full set of suffixes as intended because it implicitly demands recursive dependency that violates the problem constraints.

2. **Kleene Star (e*):**

Peer Review ID: 311012197 — enter this when you fill out your peer evaluation via gradescope

For this, e* we need to match 0 or more repetitions which can lead to infinite possibilities of suffixes.
This can moreover happen for self-matching patterns like the (.”a”)* on the string such as “aaaa”.

Thus when we attempt to bring up the rule:

Base Case: ‘ e* matches s leaving {s}

One or more occurrences:

$$\frac{\text{‘}e \text{ matches } s \text{ leaving } S, \quad \text{‘}e^* \text{ matches } s' \text{ leaving } S', \forall s' \in S}{\text{‘}e^* \text{ matches } s \text{ leaving } \{s\} \cup S'}$$

This rule is clearly not allowed. This is because, the construction of the suffix set requires recursive and potentially infinite applications of e, producing results that must themselves be further evaluated through e*. This requirement can't be satisfied with just a finite and fixed hypothesis, as the process of dealing with recursive matching is inherently unbounded.

Exercise 3F-3. Equivalence [7 points].

Claim: e1 ~ e2 is undecidable

The equivalence relation e1~e2 asks whether two regular expressions e1 and e2 behave exactly the same for every string. In other words, we want to check if they leave behind the same set of suffixes for all strings in the language. This is basically saying that, no matter what string you give them, both regular expressions should match the same suffixes every time.

The problem of figuring out whether two regular expressions are equivalent in this way is actually **undecidable**. This is because:

1. **Trying to Reduce to halting problem:**
   We know that the halting problem is undecidable. The halting problem asks whether a given program P will stop (halt) or keep running forever for a particular input. This is something we can't decide for every program, and we can use this fact to show that checking whether two regular expressions are equivalent is also undecidable.

2. **Encoding the Halting Problem in Regular Expressions**:

   To prove the undecidability, I create two regular expressions e1 and e2 that depending on the behavior of a program P:

- e1 matches strings if and only if the program P halts.
- e2 represents a regular expression that matches strings if and only if the program P does not halt.

By constructing these two regular expressions, we essentially encode the halting behavior of the program into regular expressions.

If we could decide whether e1~e2, we could see and check whether P halts or not, which is exactly the halting problem.

Example: Suppose program P writes the string "halt" if it halts and writes nothing if it does not halt.
We define:

- e1="halt" (matches the string "halt" when P halts).
- e2="" (matches nothing when P does not halt).

Now, if we could decide whether e1~e2, we could answer whether P halts:

- If e1~e2 holds, it means both regular expressions behave the same, so they match the same set of suffixes for every string. This would imply that *P* halts.
- If e1~e2 does not hold, it means the two regular expressions match different sets of suffixes, implying that P does not halt.

So, this reduction shows that if we could decide whether two regular expressions are equivalent, we'd also be able to solve the halting problem. But since the halting problem is undecidable, the equivalence of regular expressions is also undecidable

Exercise 3F-4. SAT Solving [6 points].

The last two tests are 35 and 36:

(x > y) && (y > z) && (z = 10) && (x < 12)

(x > y) && (y > z) && (z = 10) && (x < 13)

The last two tests take longer because of the complexity of the constraints. These tests involve both equality (z = 10) and inequalities (x > y, y > z), which makes things more complicated for the solver. The solver needs to handle not just the regular Boolean logic (like true/false conditions) but also the arithmetic constraints. When it faces these, it has to spend more time checking all the possibilities to find a solution. The part of the solver that checks the values for x, y, and z might not be very efficient, especially with all the dependencies between the variables.

One area that could be improved is the make_blocking_clause function. This function is used when a conflict happens, and it tries to block the set of variables that caused the issue. Right now, it checks each variable one by one, which can be slow and take up a lot of memory, especially for large or complex problems. To make this faster, we could use better algorithms or data structures to find the minimal conflict set, like the methods used in conflict-directed clause learning (CDCL).