Exercise 3F-1. Regular Expression, Large-Step [10 points]. Regular Expressions are commonly used as abstractions for string matching. Here is an abstract grammar for regular expressions:

e	$::=$	$"\mathtt{x}"$	singleton — matches the character $\hat{\mathtt{x}}$
	$\mid$	empty	skip — matches the empty string
	$\mid$	$e_1\ e_2$	concatenation — matches e_1 followed by e_2
	$\mid$	$e_1 \mid e_2$	or — matches e_1 or e_2
	$\mid$	$e*$	Kleene star — matches 0 or more occurrence of e
	$\mid$	$.$	matches any single character
	$\mid$	$["\mathtt{x}" - "\mathtt{y}"]$	matches any character between $\hat{\mathtt{x}}$ and $\hat{\mathtt{y}}$ inclusive
	$\mid$	$e+$	matches 1 or more occurrences of e
	$\mid$	$e?$	matches 0 or 1 occurrence of e

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	$::=$	nil	empty string
	$\mid$	$"\mathtt{x}" :: s$	string with first character $\hat{\mathtt{x}}$ and other characters s

We write $"\mathtt{bye}"$ as shorthand for $"\mathtt{b}" :: "\mathtt{y}" :: "\mathtt{e}" :: \mathsf{nil}$. This exercise requires you to give large-step operational semantics rules of inference related to regular expressions matching strings. We introduce a judgment:

$$\vdash e \ \mathsf{matches}\ s\ \mathsf{leaving}\ s'$$

The interpretation of the judgment is that the regular expression e matches some prefix of the string s, leaving the suffix s' unmatched. If $s' = \mathsf{nil}$ then r matched s exactly. Examples:

$$\vdash "\mathtt{h}"("\mathtt{e}"+)\ \mathsf{matches}\ "\mathtt{hello}"\ \mathsf{leaving}\ "\mathtt{llo}"$$

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

$$\begin{aligned}
&\vdash ("\mathtt{h}" \mid "\mathtt{e}")*\ \mathsf{matches}\ "\mathtt{hello}"\ \mathsf{leaving}\ && "\mathtt{ello}" \\
&\vdash ("\mathtt{h}" \mid "\mathtt{e}")*\ \mathsf{matches}\ "\mathtt{hello}"\ \mathsf{leaving}\ && "\mathtt{hello}" \\
&\vdash ("\mathtt{h}" \mid "\mathtt{e}")*\ \mathsf{matches}\ "\mathtt{hello}"\ \mathsf{leaving}\ && "\mathtt{llo}"
\end{aligned}$$

Here are two rules of inference:

$$\frac{s = "\mathtt{x}" :: s'}{\vdash "\mathtt{x}"\ \mathsf{matches}\ s\ \mathsf{leaving}\ s'} \qquad \frac{}{\vdash \mathsf{empty}\ \mathsf{matches}\ s\ \mathsf{leaving}\ s}$$

Give large-step operational semantics rules of inference for the other three primal regular expressions.

Solution:

1. $e_1\ e_2$

$$\frac{\vdash e1 \text{ matches } s \text{ leaving } s_0 \quad \vdash e2 \text{ matches } s_0 \text{ leaving } s'}{\vdash e1\ e2 \text{ matches } s \text{ leaving } s'}$$

2. $e_1 \mid e_2$
There are 2 cases, either e1 matches s or e2 matches s:

$$\frac{\vdash e1 \text{ matches } s \text{ leaving } s'}{\vdash \text{e1}|\text{e2 matches } s \text{ leaving } s'} \qquad \frac{\vdash e2 \text{ matches } s \text{ leaving } s'}{\vdash \text{e1}|\text{e2 matches } s \text{ leaving } s'}$$

3. $e*$
The first rule covers the base case where e* matches 0 occurrences of s. The second rule covers the case where e matches part of s.

$$\frac{}{\vdash e* \text{ matches } s \text{ leaving } s} \qquad \frac{\vdash e \text{ matches } s \text{ leaving } s_0 \quad \vdash e* \text{ matches } s_0 \text{ leaving } s'}{\vdash e* \text{ matches } s \text{ leaving } s'}$$

Peer Review ID: 310972887 — enter this when you fill out your peer evaluation via gradescope

Exercise 3F-2. Regular Expression and Sets [5 points]. We want to update our operational semantics for regular expressions to capture multiple suffices. We want our new operational semantics to be deterministic — it return the set of all possible answers from the single-answer operational semantics above. We introduce a new judgment:

$$\vdash e \text{ matches } s \text{ leaving } S$$

And use rules of inference like the following:

$$\frac{}{\vdash \text{"x" matches } s \text{ leaving } \{s' \mid s = \text{"x"} :: s'\}} \qquad \frac{}{\vdash \text{empty matches } s \text{ leaving } \{s\}}$$

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } S'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } S \cup S'}$$

You must do one of the following:

- *either* give operational semantics rules of inference for $e*$ and e_1e_2. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash e \text{ matches } x \text{ leaving } y\}$. Each inference rules 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 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".

Solution:
It cannot be done correctly, because it is impossible to create sound and complete rules without placing a derivation inside a set.

Attempted rule for $e1$ $e2$:

$$\frac{\vdash e1 \text{ matches } s \text{ leaving } S1 \quad \vdash e2 \text{ matches } s \text{ leaving } S2}{\vdash e1\ e2 \text{ matches } s \text{ leaving } S1 \cup S2}$$

This rule is unsound because S2 is not a valid result of matching, since e2 has to match S1, but it is impossible to show that without using a derivation in a set.

Attempted rule for $e*$:

$$\frac{\vdash e \text{ matches } s \text{ leaving } S}{\vdash e* \text{ matches } s \text{ leaving } \{s\} \cup S}$$

This rule is incomplete because it only covers the cases where e matches 0 or 1 times, but not multiple. Without derivations inside set constructors, it is impossible to capture all possible suffixes.

Exercise 3F-3. Equivalence [7 points]. In the class notes (usually marked as "optional material" for the lecture component of the class but relevant for this question) we defined an equivalence relation $c_1 \sim c_2$ for IMP commands. Computing equivalence turned out to be undecideable: $c \sim c$ iff c halts. We can define a similar equivalence relation for regular expressions: $e_1 \sim e_2$ iff $\forall s \in S.\ \vdash e_1 \textsf{ matches } s \textsf{ leaving } S_1 \wedge \vdash e_2 \textsf{ matches } s \textsf{ leaving } S_2 \implies S_1 = S_2$ (note that we are using an "updated" operational semantics that returns the set of all possible matched suffices, as in the previous problem).

You must *either* claim that $e_1 \sim e_2$ is undecidable by reducing it to the halting problem *or* explain in two or three sentences how to compute it. You may assume that I the reader is familiar with the relevant literature.

Solution: $e1 \sim e2$ is undecidable.
Let $e1$ be a regex that matches any string. Let $e2$ be a regex that matches a string that represents a non-halting program. If the program halts, then $e1$ is not equivalent to $e2$ because $e1$ will match strings representing halting programs. If the program doesn't halt, then $e1$ and $e2$ are equivalent because they match infinitely looping strings. Thus, if $e1 \sim e2$ is decidable, then you could solve the halting problem. Thus, $e1 \sim e2$ is not decidable.

Exercise 3F-4. SAT Solving [6 points]. Why do the last two included tests take such a comparatively long time? Impress me with your knowledge of DPLL(T) — feel free to use information from the assigned reading or related papers, not just from the lecture slides. I am looking for a reasonably detailed answer. Include a discussion of which single module you would rewrite first to improve performance, as well as how you would change that module.

Potential bonus point: The provided code contains at least one fairly egregious defect. Comment.

Solution: The CNFs of the last two test cases do not have unit clauses that can be assigned right away, so they need to use the bounded exhaustive search. I would rewrite the Arith module first to improve performance, which currently seems to use a brute force/bounded exhaustive search approach of trying to assign all possible integer values to each variable. It could be rewritten to use the Simplex algorithm which uses heuristics to minimize enumerating all possible values.