1 3F-1 Bookkeeping

- **0 pts** Correct

regular expressions:

e	$::=$	$\text{"}\mathtt{x}\text{"}$	singleton — matches the character $\hat{x}$
	$\mid$	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
	$\mid$	$[\text{“x”} - \text{“y”}]$	matches any character between $\hat{x}$ and $\hat{y}$ 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
	$\mid$	$\text{"}\mathtt{x}\text{"} :: s$	string with first character $\hat{x}$ and other characters s

We write $\text{"}\mathtt{bye}\text{"}$ as shorthand for $\text{"}\mathtt{b}\text{"} :: \text{"}\mathtt{y}\text{"} :: \text{"}\mathtt{e}\text{"} :: \mathsf{nil}$. 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:

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

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

$$\begin{array}{l}
\vdash (\text{"}\mathtt{h}\text{"} \mid \text{"}\mathtt{e}\text{"})*\ \mathsf{matches}\ \text{"}\mathtt{hello}\text{"}\ \mathsf{leaving}\quad \text{"}\mathtt{ello}\text{"} \\
\vdash (\text{"}\mathtt{h}\text{"} \mid \text{"}\mathtt{e}\text{"})*\ \mathsf{matches}\ \text{"}\mathtt{hello}\text{"}\ \mathsf{leaving}\quad \text{"}\mathtt{hello}\text{"} \\
\vdash (\text{"}\mathtt{h}\text{"} \mid \text{"}\mathtt{e}\text{"})*\ \mathsf{matches}\ \text{"}\mathtt{hello}\text{"}\ \mathsf{leaving}\quad \text{"}\mathtt{llo}\text{"}
\end{array}$$

Here are two rules of inference:

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

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

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } s'' \quad \vdash e_2 \textsf{ matches } s'' \textsf{ leaving } s'}{\vdash e_1\ e_2 \textsf{ matches } s \textsf{ leaving } s'}$$

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } s'}{\vdash e_1 \mid e_2 \textsf{ matches } s \textsf{ leaving } s'} \qquad \frac{\vdash e_2 \textsf{ matches } s \textsf{ leaving } s'}{\vdash e_1 \mid e_2 \textsf{ matches } s \textsf{ leaving } s'}$$

$$\frac{}{\vdash e^* \textsf{ matches } s \textsf{ leaving } s} \qquad \frac{\vdash e \textsf{ matches } s \textsf{ leaving } s'' \quad \vdash e^* \textsf{ matches } s'' \textsf{ leaving } s'}{\vdash e^* \textsf{ matches } s \textsf{ leaving } s'}$$

Exercise 3F-3. 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:

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

And use rules of inference like the following:

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

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

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: $\{x \mid \exists y.\ \vdash e \textsf{ matches } x \textsf{ leaving } y\}$. 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".

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

Peer Review ID: 68540934 — enter this when you fill out your peer evaluation via gradescope

The hypotheses of each rule of inference need to be **fixed and finite**.

It is impossible to write a rule for e_1e_2 that captures multiple suffices. Let's consider the attempt below:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s_1 \text{ leaving } S_1 \quad \dots \quad \vdash e_2 \text{ matches } s_i \text{ leaving } S_i}{\vdash e_1\ e_2 \text{ matches } s \text{ leaving } \bigcup_{s_i \in S} S_i}$$

For different s, #elements in set S is different. We need to match e_2 with all $s_i \in S$, which means the number of our hypotheses depends on conditions.

For the rule of e^*, my first attempt meets the problem of expressing both the base case and the induction step in one rule. My second attempt meets the same problem as previous, and what's more, this time we know neither the number of elements in each S or $S_{i,j}$ nor the total number of $S_{i,j}$ we have. Therefore, it is also impossible to write a rule capturing multiple suffices for e^*.

$$\frac{\vdash e \text{ matches } s \text{ leaving } S \quad \vdash e^* \text{ matches } s_i \text{ leaving } ?}{\vdash e^* \text{ matches } s \text{ leaving } ?}$$

$$\frac{\vdash e \text{ matches } s \text{ leaving } S \quad \dots \quad \vdash e \text{ matches } s_{i,j} \text{ leaving } S_{i,j} \quad \vdash e \text{ matches } s_{i+1} \text{ leaving } \{s_{i+1}\}}{\vdash e^* \text{ matches } s \text{ leaving } ?}$$

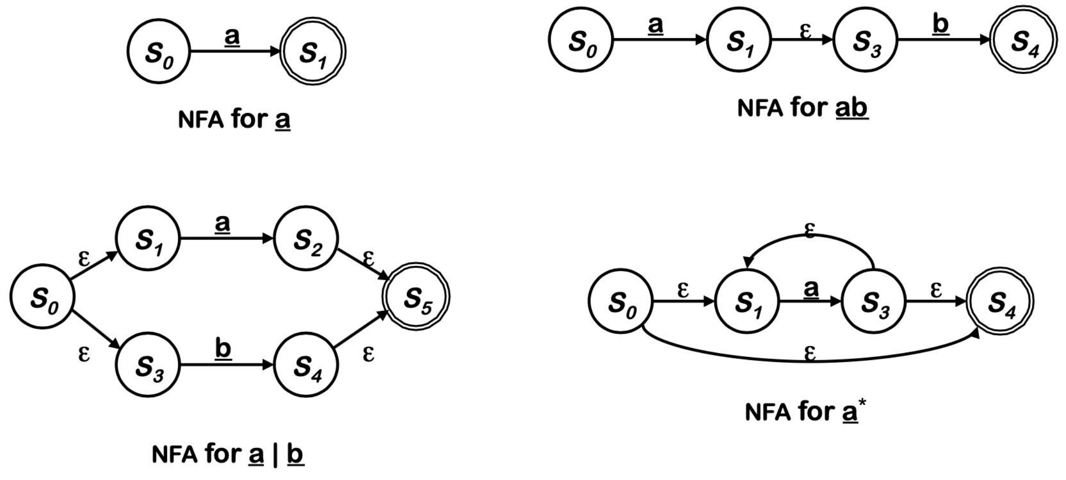
Exercise 3F-4. Equivalence [7 points]. In the class notes (usually marked as "optional material" for the lecture component of the class but relevant for this question) we defined an equivalence relation $c_1 \sim c_2$ for IMP commands. Computing equivalence turned out to be undecideable: $c \sim c$ iff c halts. We can define a similar equivalence relation for regular expressions: $e_1 \sim e_2$ iff $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \ \wedge\ \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$ (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 undecideable 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.

3 3F-3 Regular Expressions and Sets

- 0 pts Correct

$e_1 \sim e_2$ is decideable, though I cannot explain the decision process within three sentences. The idea is to convert e_1 and e_2 into finite state machines and return the equivalence of the two finite state machines.
First let me introduce two kind of finite state machines that we can create from a regular expression: NFA (nondeterministic finite automata) and DFA (deterministic finite automata). NFA can be constructed based on Thompson's Construction:



(This figure above is from EECS 483 lecture 3 slide.)
We can observe that the four patterns are actually related to four of our primary forms (excluding the empty form). Larger NFAs can be constructed by connecting the basic NFAs in Thompson's Construction according to the regular expression we want to study.
A DFA is constructed from NFA that allows one transition per input per state and no empty (in the figure, ϵ) moves. If two regular expressions have the same DFA, it means given an arbitrary input string s, the machine will execute through in the same states. That is, $S_1 = S_2$ for $\forall s \in S$ where $\vdash e_1$ matches s leaving S_1 and $\vdash e_2$ matches s leaving S_2. Therefore , the equivalence of DFA implies $e_1 \sim e_2$.
Finally, DFA equivalence problem is decidable, because it is a subset of graph isomorphism problem, which is decidable. Therefore, $e_1 \sim e_2$ is decidable.

Exercise 3C. SAT Solving. Download the Homework 3 code pack from the course web page. Update the skeletal SMT solver so that it correctly integrates the given DPLL-style CNF SAT solver with the given theory of bounded arithmetic. In particular, you must update only the Main.solve function. Your updated solver must be correct. This notably implies that it must correctly handle all of the included test cases — we use `diff` for some testing, but if you change only the listed method you should end up with the same answers as the reference.

In addition, create an example "tricky" input that can be parsed by our test harness.

Submit your `.ml` and `.input` files.

4 3F-4 Equivalence

- 0 pts Correct

Exercise 3F-5. 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.

Last two tests: The last two tests contain more inequality relationship between variables than other tests and have no unit clause or even pure literal. Thus when running these two tests, DPLL(T) can do no trick in the `SetTrue` step. All it can do is to wait for the result of the arithmetic solver. The defect (explained later) in our arithmetic solver makes the running time even longer.

Egregious defect: In `arith.ml` line 68 when finding proper assignments of variables, the algorithm starts from a fixed lower_bound (-127) and ends at either a feasible assignment or the fixed upper_bound (128). This is inefficient, because there is no need to check the assignments outside the given constraints. If an assignment doesn't exist, the current program needs to try 156 numbers before returning "None". To reduce meaningless trials, we can let the program find an interception of [lower_bound, upper_bound] and all the input constraints, and try numbers within this interception.

Submission. Turn in the formal component of the assignment as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment via the `autograder.io` website.

5 3F-5 SAT Solving

- 0 pts Correct