1 3F-1 Bookkeeping

- **0 pts** Correct

2 3F-2. Regular Expression, Large-Step

For concatenation, we simply match if both the first expression matches, and the second expression matches on the leftover string.

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

For the "or" expression, given that there is no defined preference between the two sub-expressions, we simply permit both sub-expression matches and leave the choice between them arbitrary.

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } s_1'}{\vdash e_1 | e_2 \texttt{ matches } s \texttt{ leaving } s_1'} \qquad \frac{\vdash e_2 \texttt{ matches } s \texttt{ leaving } s_2'}{\vdash e_1 | e_2 \texttt{ matches } s \texttt{ leaving } s_2'}$$

For the Kleene star, we define a sort of recursive concatenation, wherein e^* is the concatenation of e on s and e^* on the remaining s'. Our base case is the empty string, as $*$ can match any number of repetitions including zero. Like the "or" expression, the choice between these cases is nondeterministic - the number of repetitions matched is left arbitrary.

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

3 3F-3. Regular Expression and Sets

In any concatenation operation that involves sequential matching of two expressions, the latter expression's judgement depends on the remainder set of the former.

For instance, consider this inference for concatenation:

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S \qquad \forall s' \in S.\ \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S'}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } \bigcup_{s' \in S} S' \ s.t.\ \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S'}$$

I assert that this rule correctly describes the desired behavior of concatenation. However, the number of hypotheses effectively depends on the cardinality of S, as we need to make a judgement about e_2 for each possible remainder string from e_1. This violates the rule that our hypotheses must be finite and fixed. However, writing an inference without some kind of universal quantifier is obviously incomplete. Consider a rule that hypothesizes whether e_2 matches the first string in S:

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S \qquad \vdash e_2 \texttt{ matches } S_0 \texttt{ leaving } S'}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S'}$$

This holds only when $|S| = 1$. If S is empty then there is no first element, so the hypothesis is ill-formed. And if S contains multiple elements, then every element other than the first is a potential string that e_2 doesn't match, which should prelude $e_1 e_2$ from matching but is simply ignored by this rule.
In conclusion, I informally assert that in order to accurately describe regex operations that require sequence, we cannot have a fixed set of hypotheses.

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

2 3F-2. Regular Expression, Large-Step

For concatenation, we simply match if both the first expression matches, and the second expression matches on the leftover string.

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

For the "or" expression, given that there is no defined preference between the two sub-expressions, we simply permit both sub-expression matches and leave the choice between them arbitrary.

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } s_1'}{\vdash e_1 | e_2 \texttt{ matches } s \texttt{ leaving } s_1'} \qquad \frac{\vdash e_2 \texttt{ matches } s \texttt{ leaving } s_2'}{\vdash e_1 | e_2 \texttt{ matches } s \texttt{ leaving } s_2'}$$

For the Kleene star, we define a sort of recursive concatenation, wherein e^* is the concatenation of e on s and e^* on the remaining s'. Our base case is the empty string, as $*$ can match any number of repetitions including zero. Like the "or" expression, the choice between these cases is nondeterministic - the number of repetitions matched is left arbitrary.

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

3 3F-3. Regular Expression and Sets

In any concatenation operation that involves sequential matching of two expressions, the latter expression's judgement depends on the remainder set of the former.

For instance, consider this inference for concatenation:

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S \qquad \forall s' \in S.\ \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S'}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } \bigcup_{s' \in S} S' \ s.t.\ \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S'}$$

I assert that this rule correctly describes the desired behavior of concatenation. However, the number of hypotheses effectively depends on the cardinality of S, as we need to make a judgement about e_2 for each possible remainder string from e_1. This violates the rule that our hypotheses must be finite and fixed. However, writing an inference without some kind of universal quantifier is obviously incomplete. Consider a rule that hypothesizes whether e_2 matches the first string in S:

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S \qquad \vdash e_2 \texttt{ matches } S_0 \texttt{ leaving } S'}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S'}$$

This holds only when $|S| = 1$. If S is empty then there is no first element, so the hypothesis is ill-formed. And if S contains multiple elements, then every element other than the first is a potential string that e_2 doesn't match, which should prelude $e_1 e_2$ from matching but is simply ignored by this rule.
In conclusion, I informally assert that in order to accurately describe regex operations that require sequence, we cannot have a fixed set of hypotheses.

3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

4 3F-4. Equivalence

Regular expressions correspond directly to finite automata. To establish equivalence between two regular expressions e_1 and e_2, we first convert each to a deterministic finite automaton. Then, we minimize each automaton. All DFAs have a unique minimal form, so if the two minimized automata are equal (which we can establish by simultaneous graph traversal), then the two regular expressions must be equivalent.

5 3F-5. SAT Solving

The reason tests 35 and 36 take so long is due to issues in the theorem prover (the arithmetic constraint solver, `arith.ml`). From DPLL's perspective, the expression is a conjunction of four pure literals - fairly trivial to solve. However, the variables involved are anything but independent - there is only one satisfying assignment in test 36, and none in test 35. So not only is performance largely bottlenecked by the constraint solver, but due to the tight constraints on the variables and the brute-force nature of the solver, a huge amount of the search space must be explored to find the satisfying assignment (or absence of one, as in test 35).

To address this issue, the arithmetic constraint solver needs some improvements. We can't apply linear programming methods in this case, unfortunately, since constraints can be arbitrary expressions of addition and multiplication (thus, polynomial). We could certainly explore the constraints in a more direct way, however. In a manner somewhat similar to DPLL, whenever we reduce a constraint to a simple variable-to-constant equality statement (such as `z = 10`), we can substitute the value in place of that variable in all other rules. If we come across a simple variable-to-constant inequality (such as `x < 13`), we might constrain the values of that variable accordingly. If we ever come across an inconsistency (if we reduce a variable's range such that it has no valid assignments, or create a false statement by value substitution) we can immediately exit the constraint solver instead of continuing to search values. Overall, by improving the search methodology, these expensive tests would be solved in only a few iterations instead of millions.

Bonus: I noticed that the DPLL solver seems to apply pure variable elimination. In DPLL(T), pure variable elimination can lead to unnecessary inconsistencies between theories, as it assumes independent variables when theory variables are often not. This seems like a potentially major problem in some cases.

4 3F-4 Equivalence

- **0 pts** Correct

4 3F-4. Equivalence

Regular expressions correspond directly to finite automata. To establish equivalence between two regular expressions e_1 and e_2, we first convert each to a deterministic finite automaton. Then, we minimize each automaton. All DFAs have a unique minimal form, so if the two minimized automata are equal (which we can establish by simultaneous graph traversal), then the two regular expressions must be equivalent.

5 3F-5. SAT Solving

The reason tests 35 and 36 take so long is due to issues in the theorem prover (the arithmetic constraint solver, `arith.ml`). From DPLL's perspective, the expression is a conjunction of four pure literals - fairly trivial to solve. However, the variables involved are anything but independent - there is only one satisfying assignment in test 36, and none in test 35. So not only is performance largely bottlenecked by the constraint solver, but due to the tight constraints on the variables and the brute-force nature of the solver, a huge amount of the search space must be explored to find the satisfying assignment (or absence of one, as in test 35).

To address this issue, the arithmetic constraint solver needs some improvements. We can't apply linear programming methods in this case, unfortunately, since constraints can be arbitrary expressions of addition and multiplication (thus, polynomial). We could certainly explore the constraints in a more direct way, however. In a manner somewhat similar to DPLL, whenever we reduce a constraint to a simple variable-to-constant equality statement (such as `z = 10`), we can substitute the value in place of that variable in all other rules. If we come across a simple variable-to-constant inequality (such as `x < 13`), we might constrain the values of that variable accordingly. If we ever come across an inconsistency (if we reduce a variable's range such that it has no valid assignments, or create a false statement by value substitution) we can immediately exit the constraint solver instead of continuing to search values. Overall, by improving the search methodology, these expensive tests would be solved in only a few iterations instead of millions.

Bonus: I noticed that the DPLL solver seems to apply pure variable elimination. In DPLL(T), pure variable elimination can lead to unnecessary inconsistencies between theories, as it assumes independent variables when theory variables are often not. This seems like a potentially major problem in some cases.

5 3F-5 SAT Solving

- **0 pts** Correct