1 3F-1 Bookkeeping

- **0 pts** Correct

Exercise 3F-2

For the or operation, we define the following two rules of inference, one for each expression that makes up the or.

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

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

For concatenation, we define the following rule of inference.

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

Finally, for the Kleene star, we define two more rules of inference. The first corresponds to the case where we match zero occurences of e.

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

The second corresponds to the case where we match one or more occurences of e.

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

Exercise 3F-3

We claim that it is not possible to define large step operational semantics for concatenation and the Kleene star in this framework. The key problem that arises comes from the fact that both of these operations require matching on one expression on s and then matching another expression on *whatever string is left by the first expression*. However, because matching the first expression leaves (potentially) more than one suffix, it is impossible to define a correct rule of inference for either of these operations. As a demonstration, consider the following two attempted inference rules for concatenation.

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S_1 \quad (\exists s' \in S_1. \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S_2)}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S_2}$$

This rule of inference is unsound because it is not deterministic. For example if $e_1 =$ `'a' | 'aa'`, $e_2 =$ `'a'` and $s =$ `'aaa'`, then this rule of inference would allow us to conclude $\vdash e_1 e_2$ `matches` s `leaving` {`'a'`} and $\vdash e_1 e_2$ `matches` s `leaving` {`''`} but our goal in this extended definition was for the operational semantics to be deterministic. Additionally we could try the following rule.

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S_1 \quad (\forall s' \in S_1. \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S_2)}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S_2}$$

But again, this is incorrect; this time it is incomplete. Taking e_1, e_2 and s as before, this rule would make it so that there is no set S such that we can conclude $\vdash e_1 e_2$ `matches` s `leaving` S, but our intuition about regular expressions would lead us to want $S =$ {`'a'`, `''`} to suffice.

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

Exercise 3F-2

For the or operation, we define the following two rules of inference, one for each expression that makes up the or.

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

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

For concatenation, we define the following rule of inference.

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

Finally, for the Kleene star, we define two more rules of inference. The first corresponds to the case where we match zero occurences of e.

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

The second corresponds to the case where we match one or more occurences of e.

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

Exercise 3F-3

We claim that it is not possible to define large step operational semantics for concatenation and the Kleene star in this framework. The key problem that arises comes from the fact that both of these operations require matching on one expression on s and then matching another expression on *whatever string is left by the first expression*. However, because matching the first expression leaves (potentially) more than one suffix, it is impossible to define a correct rule of inference for either of these operations. As a demonstration, consider the following two attempted inference rules for concatenation.

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S_1 \quad (\exists s' \in S_1. \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S_2)}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S_2}$$

This rule of inference is unsound because it is not deterministic. For example if $e_1 =$ `'a' | 'aa'`, $e_2 =$ `'a'` and $s =$ `'aaa'`, then this rule of inference would allow us to conclude $\vdash e_1 e_2$ `matches` s `leaving` {`'a'`} and $\vdash e_1 e_2$ `matches` s `leaving` {`''`} but our goal in this extended definition was for the operational semantics to be deterministic. Additionally we could try the following rule.

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S_1 \quad (\forall s' \in S_1. \vdash e_2 \texttt{ matches } s' \texttt{ leaving } S_2)}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S_2}$$

But again, this is incorrect; this time it is incomplete. Taking e_1, e_2 and s as before, this rule would make it so that there is no set S such that we can conclude $\vdash e_1 e_2$ `matches` s `leaving` S, but our intuition about regular expressions would lead us to want $S =$ {`'a'`, `''`} to suffice.

3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

Exercise 3F-4

We claim that determining whether $e_1 \sim e_2$ is undecidable. It suffices to demonstrate a reduction to the problem of determining whether two IMP programs are equivalent, because we have shown in class that that is undecidable. We will create a map A that converts IMP programs into regular expressions. Define A recursively by pattern matching as follows:

$$\begin{aligned} A(\texttt{skip}) &= \texttt{empty} \\ A(x \coloneqq a) &= \texttt{`x := [a];'} \\ A(c_1; c_2) &= A(c_1)A(c_2) \\ A(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) &= A(c_1)|A(c_2) \\ A(\texttt{while } b \texttt{ do } c) &= A(c)* \end{aligned}$$

Where `[a]` is meant to indicate that the expression a is written out as a string (not just the character `'a'`). Now for any IMP program c, note the following things about $A(c)$. First, the only literal strings that appear in $A(c)$ correspond to variable assignments, and otherwise $A(c)$ preserves the possible control flow paths of c. So, if $A(c)$ matches a given string, the matched portion is a valid sequence of variable assignments that could appear (in order) in an execution of c. Next, because the regular expression or operator and Kleene star are both non-deterministic, $A(c)$ will match strings that correspond to *any* possible sequence of assignments that could occur in an execution of c.

Let c_1, c_2 are IMP programs. We will show that $c_1 \approx c_2 \iff A(c_1) \sim A(c_2)$. First, if $c_1 \approx c_2$, then, for all σ, σ', $\langle c_1, \sigma\rangle \Downarrow \sigma' \iff \langle c_2, \sigma\rangle \Downarrow \sigma'$. So, if we have a string s such that $A(c_1)$ `matches` s `leaving` S_1 and $A(c_2)$ `matches` s `leaving` S_2, we wish to show that $S_1 = S_2$. Because we presuppose that they both match s, we are already guaranteed that c_1 and c_2 make assignments in the same order (when executed in states that correspond to s). Moreover, if $A(c_1)$ matches some substring of s leaving $s_1 \in S_1$, that corresponds to an execution of c_1 with some starting state σ resulting in some σ'. Because $c_1 \approx c_2$, we then have that $\langle c_2, \sigma\rangle \Downarrow \sigma'$ evaluates to the same final state. But, since s is a sequence of variable assignments, this means that $A(c_2)$ must also match s_1 because s_1 corresponds directly to the state σ' (because it corresponds to starting from σ executing the sequence of variable assignments contained in s_1). Thus we have $s_1 \in S_2$ and, since s_1 was arbitrary, we conclude $S_1 \subseteq S_2$. Then, symmetrically, we have $S_2 \subseteq S_1$ and thus $S_1 = S_2$ as desired.

Next suppose $A(c_1) \sim A(c_2)$. This means that, for any $s \in S$, if $A(c_1)$ `matches` s `leaving` S_1 and $A(c_2)$ `matches` s `leaving` S_2, then $S_1 = S_2$. Let σ, σ' be states such that $\langle c_1, \sigma\rangle \Downarrow \sigma'$. We wish to show that $\langle c_1, \sigma\rangle \Downarrow \sigma'$. Let s be the sequence of variable assignments that occur in the execution of $\langle c_1, \sigma\rangle$. By construction of $A(c_1)$, we then have that $A(c_1)$ `matches` s `leaving` S_1, and, moreover, that the empty string ε is contained in S_1. Then, because $A(c_1) \sim A(c_2)$, we get that $A(c_2)$ `matches` s `leaving` S_1. Hence there is an execution of c_2 that exactly matches the sequence of variable assignments in s. But this must precisely mean that $\langle c_2, \sigma\rangle \Downarrow \sigma'$ because σ' is exactly what is obtained by starting with σ and executing all the variable assignments in s. Thus we conclude that $c_1 \approx c_2$ as desired.

Finally, define the following reduction from the IMP program equivalence problem that uses a decision oracle for regular expression equivalence problem as a black box. Given IMP programs c_1, c_2, compute $A(c_1)$ and $A(c_2)$ and call the oracle with $(A(c_1), A(c_2))$, return

YES if and only if it does. This reduction is correct by the argument given above. Thus we have shown that determining whether regular expressions are equivalent is an undecidable problem.

Exercise 3F-5

Looking at the last two test cases, we can see that both are already in CNF form and moreover that each clause contains only one literal. This makes these two test cases fairly trivial for the DPLL portion of the SMT solver, because it can use the unit clause heuristic to conclude that every one of these literals must be true.

From this we can conclude that the performance bottleneck in running these two test cases comes from the arithmetic solver. Indeed, upon further inspection, we realize that the arithmetic solver used in this assignment simply brute forces satisfiability by enumerating over all possible (integer) values of the variables (within some arbitrary bounds) and checking whether each possible assignment satisfies. This is incredibly inefficient, having runtime $O(256^n)$ where n is the number of arithmetic variables.

In order to improve the performance of this SMT solver, we would focus our efforts on improving the arithmetic solver. One possible option would be to replace the current arithmetic solver algorithm with the simplex method (or one of its variants) discussed in class.

4 3F-4 Equivalence

- **0 pts** Correct

YES if and only if it does. This reduction is correct by the argument given above. Thus we have shown that determining whether regular expressions are equivalent is an undecidable problem.

Exercise 3F-5

Looking at the last two test cases, we can see that both are already in CNF form and moreover that each clause contains only one literal. This makes these two test cases fairly trivial for the DPLL portion of the SMT solver, because it can use the unit clause heuristic to conclude that every one of these literals must be true.

From this we can conclude that the performance bottleneck in running these two test cases comes from the arithmetic solver. Indeed, upon further inspection, we realize that the arithmetic solver used in this assignment simply brute forces satisfiability by enumerating over all possible (integer) values of the variables (within some arbitrary bounds) and checking whether each possible assignment satisfies. This is incredibly inefficient, having runtime $O(256^n)$ where n is the number of arithmetic variables.

In order to improve the performance of this SMT solver, we would focus our efforts on improving the arithmetic solver. One possible option would be to replace the current arithmetic solver algorithm with the simplex method (or one of its variants) discussed in class.

5 3F-5 SAT Solving

- **0 pts** Correct