1 3F-1 Bookkeeping

- **- 0 pts** Correct

Exercise 3F-2

The large-step rules for the other three primal regular expressions are:

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

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

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

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

Peer Review ID: 68510302 — enter this when you fill out your peer evaluation via gradescope

Exercise 3F-3

We cannot modify our regular expression operational semantics to capture multiple suffixes within our current framework. Suppose we were trying to write a rule for concatenation; $e_1\ e_2$. We could start our rule as:

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

But how would we continue it? We would need to add a condition for e_2 matching any of the strings in S', of which there could be arbitrarily many. Thus, we would need an arbitrarily large number of conditions (unless we put a derivation inside a set constructor, which we have disallowed). One bad rule would be:

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } S' \quad \vdash e_2 \textsf{ matches } S'[0] \textsf{ leaving } S}{\vdash e_1\ e_2 \textsf{ matches } s \textsf{ leaving } S}$$

This rule chooses an arbitrary string from the first matching set to use for the second matching set. However, this would fail for something like $(.*\ \textsf{empty})$ matching "a". If we have $\vdash .*$ matches "a" leaving $\{$"a"$, \textsf{nil}\}$, then we *do not* have $\vdash$ empty matches "a" leaving anything. This would erroneously say that $(.*\ \textsf{empty})$ does not match "a". Another bad rule would be:

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } S_1 \quad \vdash e_2 \textsf{ matches } s \textsf{ leaving } S_2}{\vdash e_1\ e_2 \textsf{ matches } s \textsf{ leaving } S_1 \wedge S_2}$$

This is obviously faulty since e_2 should be applied to s *after* e_1. Thus any combination of S_1, S_2 in the bottom part of the rule would be erroneous.

3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

Exercise 3F-4

Determining whether $e_1 \sim e_2$ *is* a decidable problem. It's easy to see that $e_1 \sim e_2$ by our definition $\iff L(e_1) = L(e_2)$, where we define that $L(e) = \{s : \ \vdash e \ \mathsf{matches}\ s\ \mathsf{leaving}\ S \text{ where } \mathsf{nil} \in S\}$. This is the language "accepted by" the regular expression. Kleene's Theorem says that the set of languages which are acceptable by regular expressions, non-deterministic finite automata, and deterministic finite automata are the same. So the regular expressions e_1 and e_2 can be converted to DFAs (proofs of Kleene's Theorem show how), and determining whether two DFAs accept the same language is also decidable (this is a well-known result).

4 3F-4 Equivalence

- **0 pts** Correct

Exercise 3F-5

The last two included tests take a comparatively long time to run because they consist of a list of theory clauses which all need to be true in order for the SMT problem to be satisfiable. An example (test 36) is given here:

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

There are four symbols here from the perspective of DPLL, and all must be true. As such, the efficiency of DPLL(T) in this case depends solely on the efficiency of the arithmetic constraint solver. But our arithmetic constraint solver is grossly inefficient – simply looping from -127 to 128 in each variable until the constraints are satisfied (or not). Since there is only one valid set of variable assignments that can satisfy these constraints (and none for test 37), these tests take considerable time.

In order to improve the performance on these tests, the arithmetic constraint solver should be rewritten. Instead of the current method, we could use any number of integer linear programming algorithms to solve the constraints. But even if we wanted to keep the simple "looping" design, we could:

- Set variables according to numeric equality clauses from the beginning instead of looping through all possible values: `z = 10`.
- Only loop each variable within the most constrained range given by numeric inequality clauses: `x < 13` means only loop `x` up to `13`.

At least for tests 35 and 36, this would greatly reduce the search space of the arithmetic constraint solver.

As I said, there are a number of defects in the provided code relating to performance. But an egregious defect relating to *accuracy* is the fact that it only considers variable values from -127 to 128. This would erroneously label a host of problems as unsatisfiable; just one such example is:

```
(x - y > 120) && (y > 80)
```

This is solvable with `x = 250` and `y = 90`. But any solution requires we have `x > 200`, which is out of bounds for our current constraint solver.

5 3F-5 SAT Solving

- 0 pts Correct