1 3F-1 Bookkeeping

- **0 pts** Correct

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

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

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

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

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

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

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

Here are two rules of inference:

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

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

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

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'}\ \text{or1}$$

$$\frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'}\ \text{or2}$$

$$\frac{}{\vdash e* \text{ matches } s \text{ leaving } s}\ \text{Kleene1}$$

$$\frac{\vdash e \text{ matches } s \text{ leaving } s' \quad \vdash e* \text{ matches } s' \text{ leaving } s''}{\vdash e* \text{ matches } s \text{ leaving } s''}\ \text{Kleene2}$$

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.

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

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

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'}\ \text{or1}$$

$$\frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'}\ \text{or2}$$

$$\frac{}{\vdash e* \text{ matches } s \text{ leaving } s}\ \text{Kleene1}$$

$$\frac{\vdash e \text{ matches } s \text{ leaving } s' \quad \vdash e* \text{ matches } s' \text{ leaving } s''}{\vdash e* \text{ matches } s \text{ leaving } s''}\ \text{Kleene2}$$

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".

This unfortunately can't be done with the rules in place. Without nesting derivations in set constructors, there is no way to carry on work done by a prior operation as is required in the concatenation operation. Consider the following attempted derivations:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } S \text{ leaving } S' = \{s' | \exists s \in S \vdash e_2 \text{ matches } s \text{ leaving } s'\}}{\vdash e_1\ e_2 \text{ matches } s \text{ leaving } S'}$$

The above rule for concatenation unfortunately needs to use a derivation inside of a set constructor. We need some way to build off all of the work done by the first expression string by string.

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

This rule for concatenation does not need a derivation inside of the set constructor! Unfortunately, the result will be incorrect as we are only able to check if both e1 and e2 match s, but not in succession.

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 $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \ \wedge\ \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$ (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.

We can show that $e_1 \sim e_2$ is undecidable by reducing it to the halting problem. We assume that there is a decider function for $e_1 \sim e_2$ that runs in polynomial time and terminates called $ISEQUAL(e_1, e_2)$ which returns true if $e_1 \sim e_2$ and false otherwise. We then construct the following functions:

```
helper(h):
    h();
    return "X";
```

This function takes in a program h and runs h, waiting for it to terminate, before returning the regular expression "X" which matches the character X.

3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

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".

This unfortunately can't be done with the rules in place. Without nesting derivations in set constructors, there is no way to carry on work done by a prior operation as is required in the concatenation operation. Consider the following attempted derivations:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } S \text{ leaving } S' = \{s' | \exists s \in S \vdash e_2 \text{ matches } s \text{ leaving } s'\}}{\vdash e_1\ e_2 \text{ matches } s \text{ leaving } S'}$$

The above rule for concatenation unfortunately needs to use a derivation inside of a set constructor. We need some way to build off all of the work done by the first expression string by string.

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

This rule for concatenation does not need a derivation inside of the set constructor! Unfortunately, the result will be incorrect as we are only able to check if both e1 and e2 match s, but not in succession.

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 $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \ \wedge\ \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$ (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.

We can show that $e_1 \sim e_2$ is undecidable by reducing it to the halting problem. We assume that there is a decider function for $e_1 \sim e_2$ that runs in polynomial time and terminates called $ISEQUAL(e_1, e_2)$ which returns true if $e_1 \sim e_2$ and false otherwise. We then construct the following functions:

```
helper(h):
    h();
    return "X";
```

This function takes in a program h and runs h, waiting for it to terminate, before returning the regular expression "X" which matches the character X.

haltingSolver(h):
    return ISEQUAL("X",helper(h));
Where haltingSolver returns whatever ISEQUAL returns when passed in the regular expression "X" and the output of helper(h).

We see that ISEQUAL("X",helper(h)) will return true if the result of helper(h) is an equivalent regular expression to "X". We see that this definitely be the case if helper(h) return "X" which can only happen if h halts. If h does not halt, then nothing will be returned by helper(h) which means ISEQUAL("X",helper(h)) will return false. Thus, we have constructed a solver for the halting problem for any program input h. We know this program runs in polynomial time since $e_1 \sim e_2$ is assumed to be a decidable property so we have an efficient decider for the halting problem. Unfortunately, we know the halting problem is undecidable which means the formulated haltingSolver cannot exist which means that there is no such ISEQUAL program that can decide if $e_1 \sim e_2$ meaning this property is undecidable.

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.

Submitted to Autograder!

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.

The problem with the implementation here is that the theory solver is doing a brute force search to solve equations using values from -127 to 128. The problem with this, especially when there are arithmetic expressions with multiple variables (x, y, and z), the solver will perform a back tracking search in $O(256^n)$ time where n is the number of variables present. That is absolutely ridiculous and we can do a lot of better things (probably). I would rewrite the arithmetic solver module by using a more efficient LP solver (at least one that runs in polynomial time).

4 3F-4 Equivalence

- **0 pts** Correct

haltingSolver(h):
    return ISEQUAL("X",helper(h));
Where haltingSolver returns whatever ISEQUAL returns when passed in the regular expression "X" and the output of helper(h).

We see that ISEQUAL("X",helper(h)) will return true if the result of helper(h) is an equivalent regular expression to "X". We see that this definitely be the case if helper(h) return "X" which can only happen if h halts. If h does not halt, then nothing will be returned by helper(h) which means ISEQUAL("X",helper(h)) will return false. Thus, we have constructed a solver for the halting problem for any program input h. We know this program runs in polynomial time since $e_1 \sim e_2$ is assumed to be a decidable property so we have an efficient decider for the halting problem. Unfortunately, we know the halting problem is undecidable which means the formulated haltingSolver cannot exist which means that there is no such ISEQUAL program that can decide if $e_1 \sim e_2$ meaning this property is undecidable.

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.

Submitted to Autograder!

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.

The problem with the implementation here is that the theory solver is doing a brute force search to solve equations using values from -127 to 128. The problem with this, especially when there are arithmetic expressions with multiple variables (x, y, and z), the solver will perform a back tracking search in $O(256^n)$ time where n is the number of variables present. That is absolutely ridiculous and we can do a lot of better things (probably). I would rewrite the arithmetic solver module by using a more efficient LP solver (at least one that runs in polynomial time).

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

I believe a defect here that could cause problems is the fact that the CNF solver can map the same arithmetic expressions to different temporary booleans.

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