Question assigned to the following page: 2

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.

Peer Review ID: 311005776 — enter this when you fill out your peer evaluation via gradescope

Question assigned to the following page: 2

Solution: We give the large-step operational semantics rules for our primitive regular expressions as follows:

`singleton` and `skip` are repeated from above for holisticity.

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

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

$$\frac{e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'} \text{ or-left} \qquad \frac{e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'} \text{ or-right}$$

$$\frac{\text{empty matches } s \text{ leaving } s}{\vdash e* \text{ matches } s \text{ leaving } s} \text{ kleene-star-end}$$

$$\frac{e \text{ matches } s \text{ leaving } s' \quad e* \text{ matches } s' \text{ leaving } s''}{\vdash e* \text{ matches } s \text{ leaving } s''} \text{ kleene-star-recurse}$$

Non-solution stuff: For fun, and to sort of check my understanding, here are inference rules for the last four expressions as follows:

$$\frac{s = \texttt{c} :: s' \ \forall c \in \text{set of valid characters}}{\vdash . \text{ matches } s \text{ leaving } s'} \text{ any-single}$$

$$\frac{\forall c \in [\text{"}x\text{"} - \text{"}y\text{"}].\ \text{empty} \mid \texttt{c} \text{ matches } s \text{ leaving } s'}{\vdash [\text{"}x\text{"} - \text{"}y\text{"}] \text{ matches } s \text{ leaving } s'} \text{ any-between}$$

$$\frac{(e\ e*) \text{ matches } s \text{ leaving } s'}{\vdash e+ \text{ matches } s \text{ leaving } s'} \text{ one-or-more}$$

$$\frac{\text{empty} \mid e \text{ matches } s \text{ leaving } s'}{\vdash e? \text{ matches } s \text{ leaving } s'} \text{ zero-or-one}$$

Question assigned to the following page: 3

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 $e_1 e_2$. 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".

Solution: We attempt to define the inference rules for `e*` and $(e_1\ \ e_2)$ as follows:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \qquad \forall s' \in S.\ \vdash e_2 \text{ matches } s' \text{ leaving } S'_{s'}}{\vdash e_1\ e_2 \text{ matches } s \text{ leaving } \bigcup_{s'} S'_{s'}} \text{ concat}$$

$$\frac{\vdash \texttt{empty} \mid e \text{ matches } s \text{ leaving } S \quad \forall s' \in S.\ \vdash \texttt{empty} \mid e* \text{ matches } s' \text{ leaving } S'_{s'}}{\vdash e* \text{ matches } s \text{ leaving } \bigcup_{s'} S'_{s'}} \text{ kleene-star}$$

This fails however when considering how the set S could be infinite (consider $.*$ which yields an uncountably infinite set of strings). Thus, when iterating over all strings in S in the latter premise in both inference rules, we risk never completing an iteration. In a sense, this corresponds to having infinite hypotheses.

Furthermore, $\forall s \in S \vdash e \text{ matches } s \text{ leaving } S'_s$ is equivalent to writing $\{s' \mid \forall s \in S \ \vdash e \text{ matches } s \text{ leaving } s'\}$, which includes a derivation inside a set constructor. This only further invalidates these derivations.

Question assigned to the following page: 4

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.

Solution: Let us define the languages for e_1 and e_2 as L_1 and L_2 respectively. We then check whether $((L_1 \cup \overline{L}_2) \cap (\overline{L}_1 \cup L_2)) = \emptyset$ through defining a DFA, D', of which accepts only strings of this form (DFAs are capable of handling set intersection, union, and complements). If the language of $D' = \emptyset$, then it follows that $e_1 \sim e_2$ (this is possible as checking whether the language of a DFA is empty is decidable).

Non-solution stuff: The following are for myself for developing a holistic understanding. These need not to be counted towards my solution, since it would likely be far too many sentences and this could be wrong...

We could also find the minimum DFA for both the languages of e_1 and e_2, let's call these minimum DFAs D_1 and D_2 respectively. It is a fact that the minimum DFA for a language is **unique**. Thus it follows that if $D_1 \cong D_2$ then $e_1 \sim e_2$, that is, if D_1 is isomorphic to D_2, they are indeed the same DFA, and by definition of a DFA, have the same language.

We could also reference the pumping lemma and find a point p where the strings accepted by both regular expressions have seemingly repeated themselves. From here, we simply check whether the string xyz is identical in both regular expressions, where x and z are prefixes and suffices, respectively, and y is a repeated substring. Or we could naively check all strings of up to length p, and compare whether both are accepted by each regular expression.

Note: All these methods work because saying we leave sets from matching strings with a regular expression seemingly removes the non-deterministic nature we had aforementioned, thus we can represent these as DFAs.

No questions assigned to the following page.

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.

Question assigned to the following page: 5

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.

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

Solution: This implementation of SAT solving with DPLL(T) has a significant pitfall when it comes to checking for satisfying assignments to numerical variables in arithmetic expressions. While we bound the range of variables between -127 and 128, we ultimately end up exhaustively checking all possible variable assignments, only performing early stopping when the criterion of a valid arithmetic model being found in `arith.arith` has been met. This means we exhaustively check *all* 256 values within the range $[-127, 128]$ inclusive, only stopping when we find a valid assignment of variables satisfying our CNF expression. Thus, `arith.arith` proves to be the bottleneck of our algorithm, as it has a runtime which scales *exponentially* with the number of variables defined in our CNF. To be precise, this algorithm has a runtime of $O(256^n)$ where n is the number of variables which exist in our CNF. This exhausts all possible variable-value combinations in the worst-case scenario (the worst-case being our CNF is unsatisfiable, thus never reaching the early stopping criterion).

Currently, we can see the adverse negative consequences this implementation is imposing on our runtime in test cases 35 and 36, where we have 3 variables defined in our CNF. Indeed test case 35 takes the longest of any to run, as it is unsatisfiable and represents the worst-case. When we try to add a fourth variable to our CNF, the runtime becomes seemingly far too great, and we wait for eternities.

This bottleneck in our implementation represents an egregious defect, and using a method such as the Simplex algorithm would greatly reduce the worst-case runtime of our algorithm. We could also use heuristical approaches, such as pruning clauses and variable assignments to avoid repeated and unnecessary variable-assignment combinations (eg. given $x + y = 10$ and assigning $x = 3$, we can convert this to $y = 7$ and avoid exhaustively checking all 256 variable assignments).