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.

Answer:

1. $$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s' \qquad \vdash e_2 \text{ matches } s' \text{ leaving } s''}{\vdash e_1 e_2 \text{ matches } s \text{ leaving } s''}$$

2. $$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1 | e_2 \text{ matches } s \text{ leaving } s'}$$

3. $$\frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1 | e_2 \text{ matches } s \text{ leaving } s'}$$

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

5. $$\frac{\vdash e \text{ matches } s \text{ leaving } s' \qquad \vdash e* \text{ matches } s' \text{ leaving } s''}{\vdash e* \text{ matches } s \text{ leaving } s''}$$

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

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 frame-work. 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".

Answer:

We think it cannot be done correctly in the given frame-work. Both cannot be express by finite and fixed set of hypothesis. We present following attempted:

1. $$\frac{}{\vdash e* \text{ matches } s \text{ leaving } \{s\}}$$
2. $$\frac{\vdash e \text{ matches } s \text{ leaving } S' \qquad \forall s_i \in S',\ \vdash e* \text{ matches } s_i \text{ leaving } S_i}{\vdash e* \text{ matches } s \text{ leaving } \{s\} \cup_{i=1}^{|S'|} S_i}$$
3. $$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S' \qquad \forall s_i \in S',\ \vdash e_2 \text{ matches } s_i \text{ leaving } S_i}{\vdash e_1e_2 \text{ matches } s \text{ leaving } \cup_{i=1}^{|S'|} S_i}$$

Notice that S' is not a hypothesis. S' is a variable inside a hypothesis, which is not finite and not fixed, which means we may have infinite and not fixed hypotheses in the "$\forall$"' part. We cannot write down an inference rule that has a countably or uncountably infinite number of hypotheses. So it cannot be done correctly in the given frame-work.

3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

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

Answer:

$e_1 \sim e_2$ is decidable.

$e_1 \sim e_2$ iff their expressions are equivalent. And we have decidable algorithm that can compare whether two regular expressions are equivalent.

Algorithm:

`Input:` e_1, e_2

1. Transform e_1 and e_2 to standard Regular expression r_1 and r_2
2. Use Thompson's construction algorithm to transform r_1 and r_2 to Nondeterministic finite automaton nfa_1 and nfa_2
3. Use subset construction algorithm to transform nfa_1 and nfa_2 to Deterministic finite automaton dfa_1 and dfa_2
4. Reduce dfa_1 and dfa_2 to minimal DFA(canonical form) $minDFA_1$ and $minDFA_2$
5. (a) If $minDFA_1$ and $minDFA_2$ are equivlant, output $e_1 \sim e_2$
   (b) Else, output $e_1 \not\sim e_2$

Decidable:

We can easily transform our version of regular expression to the standard expression with $O(n)$ times. It's already proved in PL area that regular expression can be transformed to a NFA, NFA can be transformed to DFA, and DFA can be transformed to minimal DFA with decidable program. And there is efficient program that can compare minimum DFAs. So the total program is decidable.

Correctness:

For an arbitry $s = c_1 c_2 \dots c_n$ of length n, where c_1 to c_n are n chars, we define two sets S^+ and S^-:

$$S^+ = \{\mathsf{nil}, c_1, c_1 c_2, \cdots, c_1 \dots c_{n-1}, c_1 \dots c_n\}$$
$$S^- = \{c_1 \dots c_n, c_2 \dots c_n, c_3 \dots c_n, \cdots, c_n, \mathsf{nil}\}$$

$\forall i$, we have $S^+[i]S^-[i] = s$. We also define $\ominus$ as:

$$s \ominus S' = \{s'' | s's'' = s, s' \in S', S' \subseteq S^+, s'' \in S^-\}$$

For a regular expression r, we call R as the set of language that r define. Assume e_1 and e_2's corresponding regular expression are mapped to set R_1 and R_2. Thus, we can write the following two expression:

$$S_1 = s \ominus (R_1 \cap S^+)$$
$$S_2 = s \ominus (R_2 \cap S^+)$$

Thus, we have

$$R_1 = R_2 \iff \forall s, S_1 = S_2 \iff e_1 \sim e_2$$

Thus, to decide whether $e_1 \sim e_2$, we only need to tell whether $R_1 = R_2$, which is just what our algorithm do. Hence, it is correct.

4 3F-4 Equivalence

- **0 pts** Correct

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.

Answer:

I think one bottleneck of the last two included tests is in `arith.ml`, line `68-71`. The given program uses a brute-force algorithm to test every point in the whole variable space, which will have $O(256^n)$ time complexity if the number of variables is n. In `test-35` and `test-36`, the function `consider` will be called about $2^{31} \sim 2^{32}$ times. In some more efficient DPLL(X) engines, we usually use a simple, minimal interface to solve for arithmetic constraint solver. For example, we can treat it as a linear programming problem given a virtual task of maximizing the sum of all variables. Then we can apply a simplex algorithm that can greatly reduce average running time.

Another defect I found is the `dpll.ml`. In line `138-149`, we notice that the given version of dpll will do a recursive unit propagation once and do a recursive pure variable elimination. And then it will try to guess one symbol, like:

`dpll_sat` *clauses model*:

1. *clauses' model'* = `unit_propagation` *clauses model*
2. *clauses" model"* = `pure_variable_elimination` *clauses' model'*
3. return `dpll_sat` *clause"[c0=true] model"* or `dpll_sat` *clause"[c0=false] model"*

But we could do the first two step as a loop, like:

`dpll_sat` *clauses model*:

1. `while` *true*
   (a) *clauses' model'* = `unit_propagation` *clauses model*
   (b) *clauses" model"* = `pure_variable_elimination` *clauses' model'*
   (c) `if` *clauses* == *clauses"* `break else` *clauses* = *clauses"*, *model* = *model"*
2. return `dpll_sat` *clause"[c0=true] model"* or `dpll_sat` *clause"[c0=false] model"*

5 3F-5 SAT Solving

- **0 pts** Correct