Assignment 12: Regular Expression Derivatives

Due: 1:15pm, Tuesday, December 7

- Emails about this assignment should be directed to `cs81help@cs.hmc.edu`.
- Grutor office hours in Platt are Sundays **7–10pm** and Mondays 9–11pm; Prof. Stone's office hours in Olin 1251 are MW 4–5pm and TR 3–4pm and by appointment.
- The usual collaboration rules apply. You may *discuss* an exercise with any other student(s) currently taking CS 81 as long as:
  - You contribute equally;
  - You come away from this discussion only with *understanding in your head* — no written materials or computer notes may be retained;
  - Your submission is authored solely by you, on a separate occasion.
- You should refer only to materials from this semester of CS 81 (lecture notes, handouts, textbooks, grutors, profs, etc.).
- Bring a writeup/printout to class on the due date. Illegible answers will get no credit.
- Make sure your submission includes your name!

Derivatives of a Language

Given a language L and a symbol $a \in \Sigma^*$, we defined

$$L_a := \{ y \in \Sigma^* \mid ay \in L \}.$$

More generally for a finite word $w = a_1 \cdots a_n$, we define

$$\begin{aligned} L_w :=& \ (\cdots((L_{a_1})_{a_2})_{a_3}\cdots)_{a_n} \\ =& \ \{ y \in \Sigma^* \mid a_1 \cdots a_n y \in L \}. \end{aligned}$$

In class we called L_w an "intrinisic state" of L. Another term sometimes used for L_w is "*the derivative of* L *with respect to* w." As mentioned in class, a regular language L has only finitely many derivatives. That is, $\{ L_w \mid w \in \Sigma^* \}$ is a *finite* collection; although there are infinitely many different w's, lots of different w's will yield the same L_w.

Derivatives of a Regular Expression

Given a regular expression R whose language is L (that is, $L = L(R)$), for any $a \in \Sigma$ we can directly compute a regular expression whose language is L_a. This regular expression, *the derivative of* R *with respect to* a, is written $\partial_a R$ and satisfies $L(\partial_a R) = (L(R))_a$.

Our core language of regular expressions can be specified by the grammar:

$$\begin{array}{rll} R ::= & \emptyset & \text{Empty set} \\ | & \epsilon & \text{Empty string} \\ | & \sigma & \text{Single character, with } \sigma \in \Sigma \\ | & R\,R & \text{Concatenation} \\ | & R \mid R & \text{Alterative} \\ | & R^* & \text{Iteration} \end{array}$$

Since this is an inductive definition of regular expressions, we can define the derivative of a regular expression with respect to the symbol a inductively/recursively:

$$\begin{array}{rcl} \partial_a\,\emptyset & := & \emptyset \\ \partial_a\,\epsilon & := & \emptyset \\ \partial_a\,\sigma & := & \begin{cases} \epsilon & \text{if } \sigma = a \\ \emptyset & \text{if } \sigma \neq a \end{cases} \\ \partial_a\,(R\,R') & := & (\partial_a R)\,R' \mid \nu(R)\,(\partial_a R') \\ \partial_a\,(R \mid R') & := & \partial_a R \mid \partial_a R' \\ \partial_a\,(R^*) & := & (\partial_a R)\,R^* \end{array}$$

where

$$\nu(R) := \begin{cases} \epsilon & \text{if } \epsilon \in L(R) \\ \emptyset & \text{otherwise} \end{cases}$$

Thus, for example, we have

$$\partial_a\,(ab)^* = (\partial_a\,(ab))\,(ab)^* = ((\partial_a\,a)\,b \mid \nu(a)\,(\partial_a b))\,(ab)^* = (\epsilon b \mid \emptyset\emptyset)\,(ab)^*$$

which we can easily simplify to $b(ab)^*$. And indeed, if you take any string matching $(ab)^*$ that starts with a, once you pull off the initial a, the remaining string will match $b(ab)^*$.

As we did for languages, we can define the derivative of a regular expression with respect to a string $w = a_1 \cdots a_n$:

$$\partial_w R \quad := \quad \partial_{a_n}(\cdots \partial_{a_3}(\partial_{a_2}\,(\partial_{a_1} R)) \cdots)$$

So, for example, $\partial_{ab}\,(ab)^* = \partial_b(\partial_a\,(ab)^*) = \partial_b\,(b(ab)^*)$ which, when computed and simplified, is simply $(ab)^*$. And indeed, if you take any string matching $(ab)^*$ that starts with ab, once you pull off the initial ab the remainder matches $(ab)^*$.

Exercises

1. The description of derivatives doesn't say how the ν function is to be computed. One possibility would use a function $\mathsf{nullable}$ on regular expressions such that with $\mathsf{nullable}(R) = \mathsf{true}$ if R matches the empty string and $\mathsf{nullable}(R) = \mathsf{false}$ otherwise.

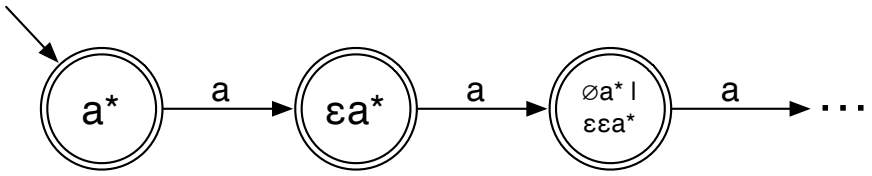
   Give a definition for the $\mathsf{nullable}$ function, defined by recursion/induction.

2. In class, we saw how we could turn the regular expression into a DFA by first turning it into an NFA, and then using the subset construction. It is less well-known that regular-expression derivatives give us a method for producing a DFA directly, without requiring an intermediate NFA.

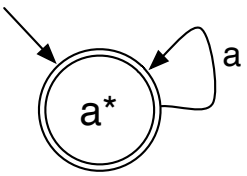
   Given a regular expression R, we can create a state machine as follows:

   - Each state is labeled by a regular expression; the start state is labeled R, and all other states are labeled with derivatives of R.
   - For each state S, and each symbol $a \in \Sigma$, add an edge labeled a from S to $\partial_a S$.
   - A state is accepting if its regular expression matches the empty string.

   One might worry that the result will be an infinite state machine. if there are infinitely many different derivatives. For example consider the alphabet $\Sigma = \{a\}$ and the regular expression a^*. According to the definition above, $\partial_a\, a^* = \epsilon a^*$, $\partial_a\, (\epsilon a^*) = \emptyset a^* \mid \epsilon\epsilon a^*$, and so on:

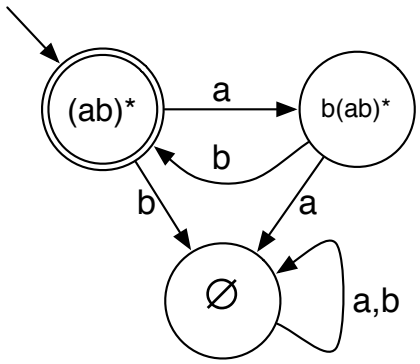


   But it is very easy to see that $\partial_a\, a^* = \epsilon a^*$ is equivalent to a^*, and so we can simply turn the edge from a^* to $\partial_a\, a^*$ into a self-loop:



   and we're done.

As another example, by using derivatives and some simple regular-expression equalities we can produce a DFA for $(ab)^*$ over the alphabet $\Sigma = \{a, b\}$:



In fact, Brzozowski (1964) showed that as long as we use some basic regular-expression equalities ($R \mid R = R$ and $R \mid S = S \mid R$ and $R \mid (S \mid T) = (R \mid S) \mid T$), we will encounter only finitely many different derivatives in the construction of the automaton. Of course, in practice there are other equalities that we should take advantage of: $\emptyset R = R\emptyset = \emptyset$, $\epsilon R = R\epsilon = R$, $(RS)T = R(ST)$, $(R^*)^* = R^*$, and so on.

The resulting automaton is not guaranteed to be minimal (because you might end up with two states having regular expressions that are equivalent but not "obviously" so), but in practice it works quite well. (The subset construction does not guarantee a minimal DFA either, and the derivative approach often yields a smaller DFA.)

Show the DFA that results from derivatives for the language $\Sigma = \{a, b\}$ and the regular expression $a^*(aab \mid bb^*a \mid bb)^*$.

Hint: you might want to use the abbreviation $E := aab \mid bb^*a \mid bb$ (e.g., labeling the start state a^*E^*). Don't forget to simplify regular expressions as you go along.

3. Because the regular languages are closed under other operations such as intersection, complement, set-difference, prefix, and shuffle, one might wonder why these operators don't show up in regular expressions. Occasionally they do; this gives us "extended" or "generalized" regular expressions.

   One reason why extended regular expressions are uncommon is that they don't work well with the standard DFA-construction method using NFAs. It's easy to take two NFAs and produce an NFA for their union, but there's no equally easy way to produce an NFA for their intersection. (As discussed in class, one can convert both NFAs into DFAs using the subset construction, and then create an intersection automaton using pairs of states, but this is much more complicated.) The same goes for other useful operators.

However, the derivative method works perfectly on extended regular expressions. Suppose we add the following operators (there's no standard notation, so I just made something up):

$R ::=$	$\ldots$	(all the cases above)
$\mid$	$!\,R$	Complement
$\mid$	$R \,\&\, R$	Intersection
$\mid$	$R \,\$\, R$	Shuffle (Sipser p. 89, Problem 1.42; see HW #8)

(a) Extend the definitions of ∂_a and $\mathsf{nullable}$ to handle these three new sorts of regular expression.

(b) Carefully *prove* that your new cases for ∂_a are correct. (Recall that a correct definition must satisfy $L(\partial_a R) = (L(R))_a$.)

You may assume this is part of a larger proof by induction that the definition of ∂_a is correct for *all* (extended) regular expressions. For example, when showing that your definition of $\partial_a(R \,\&\, R')$ is correct, you can inductively assume that $\partial_a R$ and $\partial_a R'$ are correct.

(c) Use derivatives to construct DFAs for the regular expression $a^* \;\$\; b^*$ (over the alphabet $\Sigma = \{a, b\}$) and for the regular expression $(ab)^* \;\$\; (cd)^*$ (over the alphabet $\Sigma = \{a, b, c, d\}$).

(d) Using these extended operators and any abbreviations we've discussed in class (character classes, R^+, `.`, etc.), give a regular expression R^{cc} for all strings that are valid C comments, one that is simpler and more obviously correct than that found in the previous homework. (Hint: what is the one string a C comment cannot have inside it?)

(e) Interestingly, if you have a string $w \in \Sigma^*$ that you want to check against a regular expression R, you don't necessarily have to compute the DFA: w takes you to an accepting state in a DFA for R if and only if $\mathsf{nullable}(\partial_w R) = \mathsf{true}$.

Show that `/***/` is a valid C comment by computing (and showing your work) $\partial_{/***/}\, R^{cc}$.

(f) Use derivatives to give a complete DFA for R^{cc}.