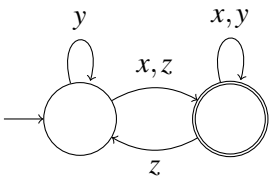
CSE 103 Computational Models
Fall 2021 Prof. Fremont HW 4

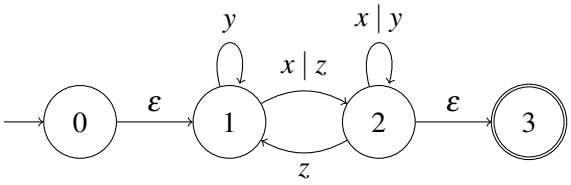
(6 questions, 220 points total)

1. (40 pts.) Converting a DFA to a regular expression

Convert the following DFA over the alphabet $\Sigma = \{x, y, z\}$ to a regular expression, showing the sequence of GNFAs you get along the way:

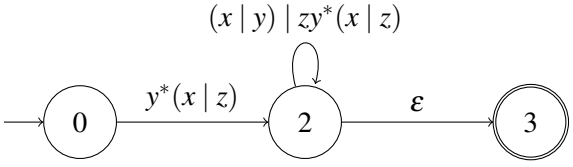


Solution: Following the procedure described in class, we first create new start and final states. We also combine the transitions from 1 to 2 on input x and z into a single transition labeled with $x \mid z$, and similarly label the self-loop on state 2 with $x \mid y$:

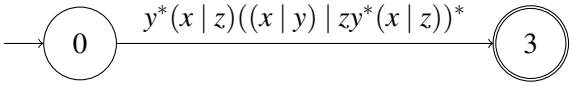


Now we need to eliminate either state 1 or state 2. Suppose we pick state 1 (we'll do the other case below). There are two "triangles" we need to consider:

- Paths going from state 0 to 1 (taking the ε transition) and then to 2 (taking the $x \mid z$ transition). Since the self-loop on state 1 is labeled by y, this set of paths is described by $\varepsilon y^*(x \mid z) = y^*(x \mid z)$. So we will create a new transition from state 0 to 2 labeled with $y^*(x \mid z)$.
- Paths going from 2 to 1 (taking the z transition) and then back to 2 (taking the $x \mid z$ transition). As above, since the self-loop on state 1 is labeled by y, this set of paths is described by $zy^*(x \mid z)$. So we will create a new transition from state 2 to itself labeled with $zy^*(x \mid z)$; since state 2 already has a self-loop, we take the union with the existing label $x \mid y$, obtaining $(x \mid y) \mid zy^*(x \mid z)$.



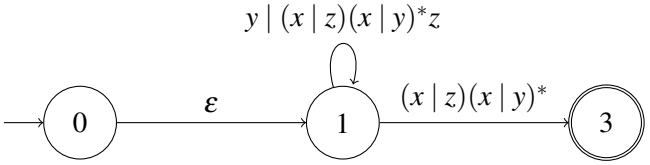
Now we eliminate state 2. There is only one "triangle", consisting of paths from state 0 to 2 and then 3. So we get:



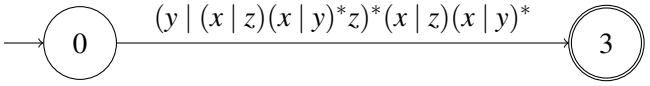
Therefore the regular expression $y^*(x \mid z)((x \mid y) \mid zy^*(x \mid z))^*$ is equivalent to the original DFA.

Alternate Solution: If instead we decide to eliminate state 2 first, we again have two triangles to consider:

- Paths going from state 1 to 2 (taking the $x \mid z$ transition) and then to 3 (taking the ε transition). Since the self-loop on state 2 is labeled by $x \mid y$, this set of paths is described by $(x \mid z)(x \mid y)^*\varepsilon = (x \mid z)(x \mid y)^*$. So we will create a new transition from state 1 to 3 labeled with $(x \mid z)(x \mid y)^*$.
- Paths going from state 1 to 2 (taking the $x \mid z$ transition) and then back to 1 (taking the z transition). Again, since the self-loop on state 2 is labeled by $x \mid y$, this set of paths is described by $(x \mid z)(x \mid y)^*z$. So we will create a new transition from state 1 to itself labeled with $(x \mid z)(x \mid y)^*z$; since state 1 already has a self-loop, we take the union with the existing label y, obtaining $y \mid (x \mid z)(x \mid y)^*z$.



Now we eliminate state 1. There is only one "triangle", consisting of paths from state 0 to 1 and then 3. So we get:



Therefore the regular expression $(y \mid (x \mid z)(x \mid y)^*z)^*(x \mid z)(x \mid y)^*$ is equivalent to the original DFA.

2. (50 pts.) Proving non-regularity using the pumping lemma

Use the pumping lemma to prove the following languages are not regular:

(a) The language L of correctly-nested parentheses (e.g. $(()()) \in L$ but $(() \notin L$).

Solution: For any given $n \geq 1$, let $w = (^n\,)^n$ (that is, n left parentheses followed by n right parentheses). Clearly $w \in L$ and $|w| \geq n$. Now for any decomposition $w = xyz$ with $y \neq \varepsilon$ and $|xy| \leq n$, the second condition implies that y consists entirely of left parentheses (since the first n symbols are left parentheses), and the first condition implies that y has at least one parenthesis. So picking $k = 2$, the string $xy^kz = xyyz$ has strictly more left parentheses than w but the same number of right parentheses (since those are all in z, which is left unchanged). Since the parentheses in w were balanced, those in xy^kz are not, so $xy^kz \notin L$. If L were regular this would contradict the pumping lemma, so L is not regular.

(b) The language $L = \{xx^R \mid x \in \Sigma^*\}$ over $\Sigma = \{0,1\}$, where x^R denotes the reversal of x (e.g. $(100)^R = 001$).

Solution: For any given $n \geq 1$, let $w = 1^n001^n$. Clearly $w \in L$ and $|w| \geq n$. Now for any decomposition $w = xyz$ with $y \neq \varepsilon$ and $|xy| \leq n$, these conditions imply that y consists of one or more 1s. Picking $k = 2$, the string $xy^kz = xyyz$ is of the form 1^m001^n with $m > n$, so $xy^kz \notin L$. If L were regular this would contradict the pumping lemma, so L is not regular.

3. (30 pts.) Big or small, not in-between

Suppose that a regular language L over Σ is recognized by a DFA with n states. Prove that if L is finite, then $|L| \leq |\Sigma^{\leq n-1}|$. (So that if the language of a DFA is finite, it's at most exponentially-large in the number of states: regular languages are either infinite or at most exponentially-large.)

(*Hint:* think about how we proved the pumping lemma.)

Solution: We'll prove the contrapositive. If $|L| > |\Sigma^{\leq n-1}|$, then L must contain some string w of length at least n, since there are only $|\Sigma^{\leq n-1}|$ words of length less than n. Since L is recognized by a DFA with n states, by the pumping lemma there then is some decomposition $w = xyz$ with $y \neq \varepsilon$ such that $xy^kz \in L$ for all $k \geq 0$. Since $y \neq \varepsilon$, the strings xy^kz are different for each value of k, and so L is infinite.

Alternate Solution: Let D be a DFA with n states recognizing L. If L is finite, then D must not have any cycles which are both reachable from the start state and able to reach an accepting state: otherwise, there would be infinitely many accepting paths and so L would be infinite. Then any accepting path in D must be a simple path (i.e. it does not repeat vertices). Since D has only n states, any simple path consists of at most n vertices. Each step along such a path follows one of $|\Sigma|$ possible outgoing edges (since each state has one transition for each symbol of Σ), so there are $|\Sigma|^{k-1} = |\Sigma^{k-1}|$ simple paths with k vertices. Therefore there are $|\Sigma^{\leq n-1}|$ simple paths with up to n vertices, so there cannot be more than this many accepting paths, and therefore $|L| \leq |\Sigma^{\leq n-1}|$.

4. (30 pts.) Working with a CFG

Consider the CFG G with terminal symbols $\Sigma = \{a, b, c\}$, nonterminal symbols $V = \{S, T, U\}$, start symbol S, and the rules

$$S \to aS \mid Sb \mid T$$
$$T \to TT \mid UU$$
$$U \to c.$$

For each of the following strings, state whether it can be derived from G, giving a derivation (a sequence of strings leading to it according to the rules) if so.

(a) ε

Solution: It cannot be derived: there are no rules allowing a nonterminal to expand into no symbols.

(b) c

Solution: It cannot be derived: notice that Us can only be produced in pairs, so the number of cs must be even.

(c) $aaccccb$

Solution: This can be derived, e.g. by $S \Rightarrow aS \Rightarrow aaS \Rightarrow aaSb \Rightarrow aaTb \Rightarrow aaTTb \Rightarrow aaTUUb \Rightarrow aaTUcb \Rightarrow aaTccb \Rightarrow aaUUccb \Rightarrow aacUccb \Rightarrow aaccccb$ (there are many possible derivations).

(d) ab

Solution: This cannot be derived, since S must ultimately expand into a T, and T must ultimately expand into a U (in both cases since otherwise we would never reach a string consisting only of terminals), so we will get at least one c.

(e) $abcc$

Solution: This cannot be derived either, since any cs must derive from T, which must come before any bs.

(f) $cccccc$

Solution: This can be derived, e.g. by $S \Rightarrow T \Rightarrow TT \Rightarrow TTT \Rightarrow UUTT \Rightarrow UUTUU \Rightarrow cUTUU \Rightarrow ccTUU \Rightarrow ccTcU \Rightarrow ccUUcU \Rightarrow cccUcU \Rightarrow cccUcc \Rightarrow cccccc$.

5. (40 pts.) Designing CFGs

Give CFGs for the following languages:

(a) With $\Sigma = \{x, y, z, +, =\}$, let L consist of all equalities between two sums of the variables x, y, z with an equal number of terms on both sides. For example the strings $x = x$ and $y + y = x + z$ are in L, but $x = y = z$ and $x = x + x$ are not.

Solution: In order to ensure that we have the same number of terms on both sides of the equality, we will create terms in pairs, adding a term simultaneously to each side using a rule similar to what we used in class for the language $0^n 1^n$:

$$S \to T + S + T \mid T = T$$
$$T \to x \mid y \mid z$$

(b) Suppose x and y are variables and $f(\cdot)$ and $g(\cdot,\cdot)$ are functions of 1 and 2 arguments respectively. With $\Sigma = \{f, g, x, y, (,), ,\}$ (where the last symbol is a comma), let L consist of all valid expressions that can be built from these symbols without having triply-nested calls to f (we don't restrict the depth of calls to g). For example, the strings x, $f(f(y))$, and $g(x, g(f(x), f(y)))$ are in L, but $g(x)$ and $f(f(g(x, f(f(y)))))$ are not (the last example having 4 layers of calls to f).

(*Hint:* First think about how to write the grammar without the nesting limit. Then you can elaborate your grammar to keep track of how many calls to f you've already made.)

Solution: Let's start by writing a grammar for these expressions ignoring the limit on calls to f. The rules encode the fact that a call to f is of the form $f(E)$, where E is an arbitrary expression, and a call to g is of the form $g(E_1, E_2)$ where E_1 and E_2 are arbitrary expressions. We also allow x and y as primitive expressions with no subexpressions:

$$E \to f(E) \mid g(E, E) \mid x \mid y$$

Now if we want to restrict the nesting depth of f, we can no longer represent all expressions with a single variable E, but need to keep track of how many calls to f we have already made. The idea is to introduce new variables E_i which can derive only expressions with at most i nested calls to f. The base case, $i = 0$, does not allow f to be used at all, and each successive case allows at most one more call to f. Implementing this idea gives the grammar:

$$E_0 \to g(E_0, E_0) \mid x \mid y$$
$$E_1 \to f(E_0) \mid g(E_1, E_1) \mid E_0$$
$$E_2 \to f(E_1) \mid g(E_2, E_2) \mid E_1$$

where the start variable is E_2. Notice that whenever we apply f, we move from E_n to E_{n-1}, so that eventually we reach E_0 and cannot nest any further calls to f.

6. (30 pts.) Ambiguity

For each of the following CFGs with $\Sigma = \{a, b\}$ and $V = \{S, T\}$, state whether it is ambiguous or unambiguous. If it is ambiguous, give an example of two different leftmost derivations for a string.

(a)

$S \to TT \mid a$
$T \to a \mid b$

Solution: There are exactly 5 strings in the language of this grammar: we can derive a directly from S, or we can go to TT and then derive aa, ab, ba, and bb. All of these strings have exactly one leftmost derivation, since from TT we have no choice but to expand the first T in order to get a *leftmost* derivation; for example, the only leftmost derivation for aa is $S \Rightarrow TT \Rightarrow aT \Rightarrow aa$. Therefore, the grammar is unambiguous.

(b)

$S \to ST \mid Sb$
$T \to a \mid b$

Solution: Notice that both productions applying to the start symbol S result in a string which contains S; therefore, it is never possible to eliminate S, and all derivable strings will contain it. In particular this means that all derivable strings will contain a nonterminal symbol, and since we define the language of a CFG to only contain strings of terminal symbols, the language of the grammar is empty. So there are no strings in the language with any leftmost derivations at all, and so the grammar is unambiguous.

(c)

$S \to TT \mid \varepsilon$
$T \to TT \mid a \mid b$

Solution: For strings of length 3 or more, we can derive them in multiple ways, by either applying the rule $T \to TT$ first to grow the string and then expanding the first T into one of the terminal symbols, or using the opposite order. For example, the string aaa has 2 distinct leftmost derivations:

$$S \Rightarrow TT \Rightarrow aT \Rightarrow aTT \Rightarrow aaT \Rightarrow aaa$$
$$S \Rightarrow TT \Rightarrow TTT \Rightarrow aTT \Rightarrow aaT \Rightarrow aaa$$

So the grammar is ambiguous.