II. Homework

CS 341 Homework 1
Basic Techniques

1. What are these sets? Write them using braces, commas, numerals, … (for infinite sets), and $\varnothing$ only.

(a) $(\{1, 3, 5\} \cup \{3, 1\}) \cap \{3, 5, 7\}$

(b) $\bigcup\{\{3\}, \{3, 5\}, \bigcap\{\{5, 7\}, \{7, 9\}\}\}$

(c) $(\{1, 2, 5\} - \{5, 7, 9\}) \cup (\{5, 7, 9\} - \{1, 2, 5\})$

(d) $2^{\{7, 8, 9\}} - 2^{\{7, 9\}}$

(e) $2^{\varnothing}$

(f) $\{x : \exists y \in N \text{ where } x = y^2\}$

(g) $\{x : x \text{ is an integer and } x^2 = 2\}$

2. Prove each of the following:

(a) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

(b) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

(c) $A \cap (A \cup B) = A$

(d) $A \cup (A \cap B) = A$

(e) $A - (B \cap C) = (A - B) \cup (A - C)$

3. Write each of the following explicitly:

(a) $\{1\} \times \{1, 2\} \times \{1, 2, 3\}$

(b) $\varnothing \times \{1, 2\}$

(c) $2^{\{1,2\}} \times \{1, 2\}$

4. Let $R = \{(a, b), (a, c), (c, d), (a, a), (b, a)\}$. What is $R \circ R$, the composition of R with itself? What is R^{-1}, the inverse of R? Is R, $R \circ R$, or R^{-1} a function?

5. What is the cardinality of each of the following sets? Justify your answer.

(a) $S = N - \{2, 3, 4\}$

(b) $S = \{\varnothing, \{\varnothing\}\}$

(c) $S = 2^{\{a, b, c\}}$

(d) $S = \{a, b, c\} \times \{1, 2, 3, 4\}$

(e) $S = \{a, b, \dots , z\} \times N$

6. Consider the chart of example relations in Section 3.2. For the first six, give an example that proves that the relation is missing each of the properties that the chart claims it is missing. For example, show that Mother-of is not reflexive, symmetric, or transitive.

7. Let A, B be two sets. If $2^A = 2^B$, must $A = B$? Prove your answer.

8. For each of the following sets, state whether or not it is a partition of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

(a) $\{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}, \{10\}\}$

(b) $\{\varnothing, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}, \{10\}\}$

(c) $\{\{1, 2\}, \{3, 4\}, \{5, 6\}, \{7, 8\}, \{9, 10\}\}$

(d) $\{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{5, 6\}, \{6, 7\}, \{7, 8\}, \{8, 9\}, \{9, 10\}\}$

9. For each of the following relations, state whether it is a partial order (that is not also total), a total order, or neither. Justify your answer.

(a) DivisibleBy, defined on the natural numbers. $(x, y) \in$ DivisibleBy iff x is evenly divisible by y. So, for example, $(9, 3) \in$ DivisibleBy but $(9, 4) \notin$ DivisibleBy.

(b) LessThanOrEqual defined on ordered pairs of natural numbers. (a, b) ≤ (x, y) iff a ≤ x or (a = x and b ≤ y). For example, (1,2) ≤ (2,1) and (1,2) ≤ (1,3).

(c) The relation defined by the following boolean matrix:

1		1	
	1	1	
		1	1
1			1

10. Are the following sets closed under the following operations? If not, what are the respective closures?

(a) The odd integers under multiplication.

(b) The positive integers under division.

(c) The negative integers under subtraction.

(d) The negative integers under multiplication.

(e) The odd length strings under concatenation.

11. What is the reflexive transitive closure R* of the relation

R = {(a, b), (a, c), (a, d), (d, c), (d, e)} Draw a directed graph representing R*.

12. For each of the following relations R, over some domain D, compute the reflexive, symmetric, transitive closure R′. Try to think of a simple descriptive name for the new relation R′. Since R′ must be an equivalence relation, describe the partition that R induces on D.

(a) Let D be the set of 50 states in the US. ∀xy, xRy iff x shares a boundary with y.

(b) Let D be the natural numbers. ∀xy, xRy iff y = x+3.

(c) Let D be the set of strings containing no symbol except a. ∀xy, xRy iff y = xa. (i.e., if y equals x concatenated with a).

13. Consider an infinite rectangular grid (like an infinite sheet of graph paper). Let S be the set of intersection points on the grid. Let each point in S be represented as a pair of (x,y) coordinates where adjacent points differ in one coordinate by exactly 1 and coordinates increase (as is standard) as you move up and to the right.

(a) Let R be the following relation on S: $\forall(x_1,y_1)(x_2,y_2)$, $(x_1,y_1)R(x_2,y_2)$ iff $x_2= x_1+1$ and $y_2=y_1+1$. Let R′ be the reflexive, symmetric, transitive closure of R. Describe in English the partition P that R′ induces on S. What is the cardinality of P?

(b) Let R be the following relation on S: $\forall(x_1,y_1)(x_2,y_2)$, $(x_1,y_1)R(x_2,y_2)$ iff $(x_2= x_1+1$ and $y_2=y_1+1)$ or $(x_2= x_1-1$ and $y_2=y_1+1)$. Let R′ be the reflexive, symmetric, transitive closure of R. Describe in English the partition P that R′ induces on S. What is the cardinality of P?

(c) Let R be the following relation on S: $\forall(x_1,y_1)(x_2,y_2)$, $(x_1,y_1)R(x_2,y_2)$ iff (x_2,y_2) is reachable from (x_1,y_1) by moving two squares in any one of the four directions and then one square in a perpendicular direction. Let R′ be the reflexive, symmetric, transitive closure of R. Describe in English the partition P that R′ induces on S. What is the cardinality of P?

14. Is the transitive closure of the symmetric closure of a binary relation necessarily reflexive? Prove it or give a counterexample.

15. Give an example of a binary relation that is not reflexive but has a transitive closure that is reflexive.

16. For each of the following functions, state whether or not it is (i) one-to-one, (ii) onto, and (iii) idempotent. Justify your answers.

(a) +: P × P → P, where P is the set of positive integers, and

+(a, b) = a + b (In other words, simply addition defined on the positive integers)

(b) X : B × B → B, where B is the set {True, False}

X(a, b) = the exclusive or of a and b

17. Consider the following set manipulation problems:

(a) Let S = {a, b}. Let T = {b, c}. List the elements of P, defined as
$P = 2^S \cap 2^T$.

(b) Let Z be the set of integers. Let $S = \{x \in Z: \exists y \in Z \text{ and } x = 2y\}$. Let $T = \{x \in Z: \exists y \in Z \text{ and } x = 2^y\}$. Let W = S – T. Describe W in English. List any five consecutive elements of W. Let X = T – S. What is X?

Solutions

1. (a) {3, 5}
(b) {3, 5, 7}
(c) {1, 2, 7, 9}
(d) {8}, {7, 8}, {8, 9}, {7, 8, 9}
(e) {∅}
(f) {0, 1, 4, 9, 25, 36…} (the perfect squares)
(g) ∅ (since the square root of 2 is not an integer)

2. (a)

$$\begin{aligned} A \cup (B \cap C) &= (B \cap C) \cup A && \text{commutativity} \\ &= (B \cup A) \cap (C \cup A) && \text{distributivity} \\ &= (A \cup B) \cap (A \cup C) && \text{commutativity} \end{aligned}$$

(b)

$$\begin{aligned} A \cap (B \cup C) &= (B \cup C) \cap A && \text{commutativity} \\ &= (B \cap A) \cup (C \cap A) && \text{distributivity} \\ &= (A \cap B) \cup (A \cap C) && \text{commutativity} \end{aligned}$$

(c)

$$\begin{aligned} A \cap (A \cup B) &= (A \cup B) \cap A && \text{commutativity} \\ &= A && \text{absorption} \end{aligned}$$

3. (a) {(1,1,1), (1,1,2), (1,1,3), (1,2,1), (1,2,2),, (1,2,3)}
(b) ∅
(c) {(∅,1), (∅,2), ({1}, 1), ({1}, 2), ({2}, 1), ({2}, 2), ({1,2}, 1), ({1,2}, 2)}

4. R ° R = {(a, a), (a, d), (a, b), (b, b), (b, c), (b, a), (a, c)}
R inverse = {(b, a), (c, a), (d, c), (a, a), (a, b)}
None of R, R ° R or R inverse is a function.

5. (a) S = {0, 1, 5, 6, 7, …}. S has the same number of elements as N. Why? Because there is a bijection between S and N: $f: S \rightarrow N$, where f(0) = 0, f(1) = 1, $\forall x \geq 5$, f(x) = x - 3. So $|S| = \aleph_0$.

(b) 2.

(c) S = all subsets of {a, b, c}. So S = {∅, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}. So |S| = 8. We could also simply have used the fact that the cardinality of the power set of a finite set of cardinality c is 2^c.

(d) S = {(a, 1), (a, 2), (a, 3), (a, 4), (b, 1), (b, 2), (b, 3), (b, 4), (c, 1), (c, 2), (c, 3), (c, 4)}. So |S| = 12. Or we could have used the fact that, for finite sets, $|A \times B| = |A| * |B|$.

(e) S = {(a, 0), (a, 1), …, (b, 0), (b, 1),…} Clearly S contains an infinite number of elements. But are there the same number of elements in S as in N, or are there more (26 times more, to be precise)? The answer is that there are the same number. $|S| = \aleph_0$. To prove this, we need a bijection from S to N. We can define this bijection as an enumeration of the elements of S:

(a, 0), (b, 0), (c, 0), … (First enumerate all 26 elements of S that have 0 as their second element)

(a, 1), (b, 1), (c, 1), … (Next enumerate all 26 elements of S that have 1 as their second element)
and so forth.

6. Mother-of: Not reflexive: Eve is not the mother of Eve (in fact, no one is her own mother).
Not symmetric: mother-of(Eve, Cain), but not Mother-of(Cain, Eve).
Not transitive: Each person has only one mother, so if Mother-of(x, y) and Mother-of(y, z), the only way to have Mother-of(x, z) would be if x and y are the same person, but we know that that's not possible since Mother-of(x, y) and no one can be the mother of herself).

Would-recognize-picture-of:
Not symmetric: W-r-p-o(Elaine, Bill Clinton), but not W-r-p-o (Bill Clinton, Elaine)
Not transitive: W r-p-o(Elaine, Bill Clinton) and W r-p-o(Bill Clinton, Bill's mom) but not W-r-p-o(Elaine, Bill's mom)

Has-ever-been-married-to: Not reflexive: No one is married to him or herself.
Not transitive: H-e-b-m-t(Dave, Sue) and H-e-b-m-t(Sue, Jeff) but not H-e-b-m-t(Dave, Jeff)

Ancestor-of: Not reflexive: not Ancestor-of(Eve, Eve) (in fact, no one is their own ancestor).
Not symmetric: Ancestor-of(Eve, Cain) but not Ancestor-of(Cain, Eve)

Hangs-out-with: Not transitive: Hangs-out-with(Bill, Monica) and Hangs-out-with(Monica, Linda Tripp), but not Hangs-out-with(Bill, Linda Tripp).

Less-than-or-equal-to: Not symmetric: $1 \leq 2$, but not $2 \leq 1$.

7. Yes, if $2^A = 2^B$, then A must equal B. Suppose it didn't. Then there is some element x that is in one set but not the other. Call the set x is in A. Then 2^A must contain {x}, which must not be in 2^B, since $x \notin B$. This would mean that $2^A \neq 2^B$, which contradicts our premise.

8. (a) yes
(b) no, since no element of a partition can be empty.
(c) no, 0 is missing
(d) no, since, each element of the original set S must appear in only one element of a partition of S.

9. (a) DivisibleBy is a partial order. $\forall x\ (x, x) \in$ DivisibleBy, so DivisibleBy is reflexive. For x to be DivisibleBy y, x must be greater than or equal to y. So the only way for both (x, y) and (y, x) to be in DivisibleBy is for x and y to be equal. Thus DivisibleBy is antisymmetric. And if x is DivisibleBy y and y is DivisibleBy z, then x is DivisibleBy z. So DivisibleBy is transitive. But DivisibleBy is not a total order. For example neither (2, 3) nor (3, 2) is in it.

(b) LessThanOrEqual defined on ordered pairs is a total order. This is easy to show by relying on the fact that ≤ for the natural numbers is a total order.

(c) This one is not a partial order at all because, although it is reflexive and antisymmetric, it is not transitive. For example, it includes (4, 1) and (1, 3), but not (4, 3).

10. (a) The odd integers are closed under multiplication. Every odd integer can be expressed as 2n+1 for some value of $n \in N$. So the product of any two odd integers can be written as (2n+1)(2m+1) for some values of n and m. Multiplying this out, we get 4(n+m) +2n + 2m +1, which we can rewrite as 2(2(n+m) + n + m) +1, which must be odd.

(b) The positive integers are not closed under division. To show that a set is not closed under an operation, it is sufficient to give one counterexample. 1/2 is not an integer. The closure of the positive integers under division is the positive rationals.

(c) The negative integers are not closed under subtraction. -2 - (-4) = 2. The closure of the negative numbers under subtraction is the integers.

(d) The negative integers are not closed under multiplication. -2 * -2 = 4. The closure of the negative numbers under multiplication is the nonzero integers. Remember that the closure is the *smallest* set that contains all the necessary elements. Since it is not possible to derive zero by multiplying two negative numbers, it must not be in the closure set.

(e) The odd length strings are not closed under concatenation. "a" || "b" = "ab", which is of length 2. The closure is the set of strings of length ≥ 2. Note that strings of length 1 are not included. Why?

11. $R^* = R \cup \{(x, x) : x \in \{a, b, c, d, e\}\} \cup \{(a, e)\}$

12. (a) The easiest way to start to solve a problem like this is to start writing down the elements of R′ and see if a pattern emerges. So we start with the elements of R: {(TX, LA), (LA, TX), (TX, NM), (NM, TX), (LA, Ark), (Ark, LA), (LA Miss), (Miss, LA) …}. To construct R′, we first add all elements of the form (x, x), so we add (TX,TX), and so forth. Then we add the elements required to establish transitivity:

(NM, TX), (TX, LA) ⇒ (NM, TX)
(TX, LA), (LA, Ark) ⇒ (TX, Ark)
(NM, TX), (TX, Ark) ⇒ (NM, Ark), and so forth.

If we continue this process, we will see that the reflexive, symmetric, transitive closure R′ relates all states except Alaska and Hawaii to each other and each of them only to themselves. So R′ can be described as relating two states if it's possible to drive from one to the other without leaving the country. The partition is:

[Alaska]
[Hawaii]
[all other 48 states]

(b) R includes, for example {(0, 3), (3, 6), (6, 9), (9, 12) …}. When we compute the transitive closure, we add, among other things {(0, 6), (0, 9), (0,12)}. Now try this starting with (1,4) and (2, 5). It's clear that ∀x,y, xR′y iff x = y (mod 3). In other words, two numbers are related iff they have the same remainder mod 3. The partition is:

[0, 3, 6, 9, 12 …]
[1, 4, 7, 10, 13 …]
[2, 5, 8, 11, 14 …]

(c) R′ relates all strings composed solely of a's to each other. So the partition is

[ε, a, aa, aaa, aaaa, …]

13. (a) Think of two points being related via R if you can get to the second one by starting at the first and moving up one square and right one square. When we add transitivity, we gain the ability to move diagonally by two squares, or three, or whatever. So P is an infinite set. Each element of P consists of the set of points that fall on an infinite diagonal line running from lower left to upper right.

(b) Now we can more upward on either diagonal. And we can move up and right followed by up and left, and so forth. The one thing we can't do is move directly up or down or right or left exactly one square. So take any given point. To visualize the points to which it is related under R′, imagine a black and white chess board where the squares correspond to points on our grid. Each point is related to all other points of the same color. Thus the cardinality of P is 2.

(c) Now every point is related to every other point. The cardinality of P is 1.

14. You might think that for all relations R on some domain D, the transitive closure of the symmetric closure of R (call it TC(SC(R))) must be reflexive because for any two elements x, y ∈ D such that (x, y) ∈ R, we'll have (x, y), (y, z) ∈ SC(R) and therefore (x, x), (y, y) ∈ TC(SC(R)). This is all true, but does not prove that for all z ∈ D, (z, z) ∈ TC(SC(R)). Why not? Suppose there is a z ∈ D such that there is no y ∈ D for which (y, z) ∈ R or (z, y) ∈ R. (If you look at the graph of R, z is an isolated vertex with no edges in or out.) Then (z, z) ∉ TC(SC(R)). So the answer is no, with R = ∅ on domain {a} as a simple counterexample: TC(SC(R)) = ∅, yet it should contain (a, a) if it were reflexive.

15. R = {(a, b), (b, a)} on domain {a, b} does the trick easily.

16. (a) (i) + is not one-to-one. For example +(1, 3) = +(2, 2) = 4.

(ii) + is not onto. There are no two positive integers that sum to 1.

(iii) + is not idempotent. +(1, 1) ≠ 1.

(b) (i) X is not one-to-one. For example True X True = False X False = False.

(ii) X is onto. Proof: True X True = False. True X False = True. In general, when the domain is a finite set, it's easy to show that a function is onto: just show one way to derive each element.

(iii) X is not idempotent. True X True = False.

17. (a) P = {∅,{b}}

(b) S is the set of even numbers. T is the set powers of 2. W is the set of even numbers that are not powers of 2. So W = { ….-6, -4, -2, 0, 6, 10, 12, 14, 18, …}. X is the set of numbers that are powers of 2 but are not even. There's only one element of X. X = {1}.

CS 341 Homework 2
Strings and Languages

1. Let $\Sigma = \{a, b\}$. Let $L_1 = \{x \in \Sigma^*: |x| < 4\}$. Let $L_2 = \{aa, aaa, aaaa\}$. List the elements in each of the following languages L:

(a) $L_3 = L_1 \cup L_2$
(b) $L_4 = L_1 \cap L_2$
(c) $L_5 = L_1 L_4$
(d) $L_6 = L_1 - L_2$

2. Consider the language $L = a^n b^n c^m$. Which of the following strings are in L?
(a) ε **(b)** ab **(c)** c **(d)** aabc **(e)** aabbcc **(f)** abbcc

3. It probably seems obvious to you that if you reverse a string, the character that was originally first becomes last. But the definition we've given doesn't say that; it says only that the character that was originally last becomes first. If we want to be able to use our intuition about what happens to the first character in a proof, we need to turn it into a theorem. Prove $\forall x,a$ where x is a string and a is a single character, $(ax)^R = x^R a$.

4. For each of the following binary functions, state whether or not it is (i) one-to-one, (ii) onto, (iii) idempotent, (iv) commutative, and (v) associative. Also (vi) state whether or not it has an identity, and, if so, what it is. Justify your answers.

(a) $\| : S \times S \rightarrow S$, where S is the set of strings of length ≥ 0
$\|(a, b) = a \| b$ (In other words, simply concatenation defined on strings)

(b) $\| : L \times L \rightarrow L$ where L is a language over some alphabet Σ
$\|(a, b) = \{w \in \Sigma^*: w = x \| y \text{ for some } x \in a \text{ and } y \in b\}$ In other words, the concatenation of two languages A and B is the set of strings that can be derived by taking a string from A and then concatenating onto it a string from B.

5. We can define a unary function F to be ***self-inverse*** iff $\forall x \in \text{Domain}(F)\ F(F(x)) = x$. The Reverse function on strings is self-inverse, for example.

(a) Give an example of a self-inverse function on the natural numbers, on sets, and on booleans.
(b) Prove that the Reverse function on strings is self-inverse.

Solutions

1. First we observe that L_1 = {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb}.

(a) L_3 = {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa}

(b) L_4 = {aa, aaa}

(c) L_5 = every way of selecting one element from L_1 followed by one element from L_4:
{εaa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, aabaa, abaaa, abbaa, baaaa, babaa, bbaaa, bbbaa} ∪
{εaaa, aaaa, baaa, aaaaa, abaaa, baaaa, bbaaa, aaaaaa, aabaaa, abaaaa, abbaaa, baaaaa, babaaa, bbaaaa, bbbaaa}. Note that we've written εaa, just to make it clear how this string was derived. It should actually be written as just aa. Also note that some elements are in both of these sets (i.e.,
there's
more than one way to derive them). Eliminating duplicates (since L is a set and thus does not contain duplicates), we get:
{aa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, aabaa, abaaa, abbaa, baaaa, babaa, bbaaa, bbbaa, aaaaaa, aabaaa, abaaaa, abbaaa, baaaaa, babaaa, bbaaaa, bbbaaa}

(d) L_6 = every string that is in L_1 but not in L_2: {ε, a, b, ab, ba, bb, aab, aba, abb, baa, bab, bba, bbb}.

2. (a) Yes. n = 0 and m = 0.

(b) Yes. n = 1 and m = 0.

(c) Yes. n = 0 and m = 1.

(d) No. There must be equal numbers of a's and b's.

(e) Yes. n = 2 and m = 2.

(f) No. There must be equal numbers of a's and b's.

3. Prove: ∀x,a where x is a string and a is a single character, $(ax)^R = x^R a$. We'll use induction on the length of x. If |x| = 0 (i.e, x = ε), then $(a\varepsilon)^R = a = \varepsilon^R a$. Next we show that if this is true for all strings of length n, then it is true for all strings of length n + 1. Consider any string x of length n + 1. Since |x| > 0, we can rewrite x as yb for some single character b.

$$\begin{aligned}
(ax)^R &= (ayb)^R && \text{Rewrite of x as yb} \\
&= b(ay)^R && \text{Definition of reversal} \\
&= b(y^R a) && \text{Induction hypothesis (since } |x| = n + 1, |y| = n) \\
&= (b\, y^R)\, a && \text{Associativity of concatenation} \\
&= x^R a && \text{Definition of reversal: If } x = yb \text{ then } x^R = by^R
\end{aligned}$$

4. (a) (i) || is not one-to-one. For example, ||(ab, c) = ||(a, bc) = abc.

(ii) || is onto. Proof: ∀s ∈ S, ||(s, ε) = s, so every element of s can be generated.

(iii) || is not idempotent. ||(a, a) ≠ a.

(iv) || is not commutative. ||(ab, cd) ≠ (cd, ab)

(v) || is associative.

(vi) || has ε as both a left and right identity.

(b) (i) || is not one to one. For example, Let Σ = {a, b, c}. ||({a}, {bc}) = {abc} = ||({ab}, {c})

(ii) || is onto. Proof: ∀L ⊆ Σ*, ||(L, {ε}) = L, so every element of s can be generated. Notice that this proof is very similar to the one we used to show that concatenation of strings is onto. Both proofs rely on the fact that ε is an identity for concatenation of strings. Given the way in which we defined concatenation of languages as the concatenation of strings drawn from the two languages, {ε} is an identity for concatenation of languages and thus it enables us to prove that all languages can be derived from the concatenation operation.

(iii) || is not idempotent. ||({a}, {a}) = {aa}

(iv) || is not commutative. ||({a}, {b}) = {ab}. But ||({b}, {a}) = {ba}.

(v) || is associative.

(vi) || has {ε} as both a left and right identity.

5. (a) Integers: F(x) = -x is self-inverse. Sets: Complement is self-inverse. Booleans: Not is self-inverse.

(b) We'll prove this by induction on the length of the string.

Base case: If |x| = 0 or 1, then $x^R = x$. So $(x^R)^R = x^R = x$.

Show that if this is true for all strings of length n, then it is true for all strings of length n + 1. Any string s of length n + 1 can be rewritten as xa for some single character a. So now we have:

$$\begin{aligned}
s^R &= a\, x^R && \text{definition of string reversal} \\
(s^R)^R &= (a\, x^R)^R && \text{substituting } a\, x^R \text{ for } s^R \\
&= (x^R)^R a && \text{by the theorem we proved above in (3)} \\
&= xa && \text{induction hypothesis} \\
&= s && \text{since xa was just a way of rewriting s}
\end{aligned}$$

CS 341 Homework 3
Languages and Regular Expressions

1. Describe in English, as briefly as possible, each of the following (in other words, describe the language defined by each regular expression):

(a) L($((a^*a)\, b) \cup b$ **)**

(b) L($(((a^*b^*)^*ab) \cup ((a^*b^*)^*ba))(b \cup a)^*$ **)**

2. Rewrite each of these regular expressions as a simpler expression representing the same set.

(a) $\emptyset^* \cup a^* \cup b^* \cup (a \cup b)^*$

(b) $((a^*b^*)^*\, (b^*a^*)^*)^*$

(c) $(a^*b)^* \cup (b^*a)^*$

3. Let $\Sigma = \{a, b\}$. Write regular expressions for the following sets:

(a) All strings in Σ^* whose number of a's is divisible by three.

(b) All strings in Σ^* with no more than three a's.

(c) All strings in Σ^* with exactly one occurrence of the substring aaa.

4. Which of the following are true? Prove your answer.

(a) $baa \in a^*b^*a^*b^*$

(b) $b^*a^* \cap a^*b^* = a^* \cup b^*$

(c) $a^*b^* \cap c^*d^* = \emptyset$

(d) $abcd \in (a(cd)^*b)^*$

5. Show that $L((a \cup b)^*) = L(a^*\, (ba^*)^*)$.

6. Consider the following:

(a) $((a \cup b) \cup (ab))^*$
(b) $(a^+\, a^nb^n)$
(c) $((ab)^*\, \emptyset)$
(d) $(((ab) \cup c)^* \cap (b \cup c^*))$
(e) $(\emptyset^* \cup (bb^*))$

(i) Which of the above are “pure” regular expressions?

(ii) For each of the above that is a regular expression, give a simplified equivalent “pure” regular expression.

(iii) Which of the above represent regular languages?

7. True - False: For all languages L1, L2, and L3

(a) $(L1L2)^* = L1^*L2^*$

(b) $(L1 \cup L2)^* = L1^* \cup L2^*$

(c) $(L1 \cup L2)\, L3 = L1\, L3 \cup L2\, L3$

(d) $(L1\, L2) \cup L3 = (L1 \cup L3)\, (L2 \cup L3)$

(e) $L1^+)^* = L1^*$

(f) $(L1^+)^+ = L1^+$

(g) $(L1^*)^+ = (L1+)^*$

(h) $L1^* = L1^+ \cup \emptyset$

(i) $(ab)^*a = a(ba)^*$

(j) $(a \cup b)^*\, b\, (a \cup b)^* = a^*\, b\, (a \cup b)^*$

(k) $[(a \cup b)^*\, b\, (a \cup b)^* \cup (a \cup b)^*\, a\, (a \cup b)^*] = (a \cup b)^*$

(l) $[(a \cup b)^*\, b\, (a \cup b)^* \cup (a \cup b)^*\, a\, (a \cup b)^*] = (a \cup b)^+$

(m) $[(a \cup b)^*\, b\, a\, (a \cup b)^* \cup a^*b^*] = (a \cup b)^*$

(n) (L1L2L3)* = L1*L2*L3*
(o) (L1* ∪ L3*) = (L1* ∪ L3*)*
(p) L1* L1 = L1+
(q) (L1 ∪ L2)* = (L2 ∪ L1)*
(r) $L1* (L2 \cup L3)^{+} = (L1* L2^{+} \cup L1* L3^{+})$
(s) ∅ L1* = ∅
(t) ∅ L1* = {ε}
(u) (L1 - L2) = (L2 - L1)
(v) ((L1 L2) ∪ (L1 L3))* = (L1 (L2 ∪ L3))*

8. Let L = {w ∈ {a, b}* : w contains bba as a substring}. Find a regular expression for {a, b}* - L.

9. Let Σ = {a,b}. For each of the following sets of strings (i.e., languages) L, first indicate which of the example strings are in the language and which are not. Then, if you can, write a concise description, in English, of the strings that are in the language.
Example strings: (1) aaabbb, (2) abab, (3) abba, (4) ε

(a) $L = \{w : \text{for some } u \in \Sigma^*, w = u^{R}u\}$

(b) L = {w : ww = www}

(c) L = {w : for some u ∈ Σ*, www = uu}

10. Write a regular expression for the language consisting of all odd integers *without* leading zeros.

11. Let Σ = {a, b}. Let L = {ε, a, b}. Let R be a relation defined on Σ* as follows: ∀xy, xRy iff y = xb. Let R′ be the reflexive, transitive closure of L under R. Let L′ = {x : ∃y ∈ L such that yR′x}. Write a regular expression for L′.

Solutions

1. (a) Any string of a's and/or b's with zero or more a's followed by a single b.
(b) Any string of a's and/or b's with at least one occurrence of ab or ba.

2. (a) ∅* = {ε}, and ε ⊆ (a ∪ b)*.
a* ⊆ (a ∪ b)*.
b* ⊆ (a ∪ b)*. So since the first three terms describe subsets of the last one, unioning them into the last one doesn't add any elements. Thus we can write simply (a ∪ b)*.

(b) To solve this one, we'll use some identities for regular expressions. We don't have time for an extensive study of such identities, but these are useful ones:

((a*b*)* (b*a*)*)* =
Using (A*B*)* = (A ∪ B)* (Both simply describe any string that is composed of elements of A and elements of B concatenated together in any order)
((a ∪ b)*(b ∪ a)*)* =
Using (A ∪ B) = (B ∪ A) (Set union is commutative)
((a ∪ b)*(a ∪ b)*)* =
Using A*A* = A*
((a ∪ b)*)* =
Using (A*)* = A*
(a ∪ b)*

(c) $(a^*b)^* \cup (b^*a)^* = (a \cup b)^*$ (In other words, all strings over {a, b}.) How do we know that? $(a^*b)^*$ is the union of ε and all strings that end in b. $(b^*a)^*$ is the union of ε and all strings that end in a. Clearly any string over {a, b} must either be empty or it must end in a or b. So we've got them all.

3. (a) The a's must come in groups of three, but of course there can be arbitrary numbers of b's everywhere. So:

$(b^*ab^*ab^*a)^*b^*$

Since the first expression has * around it, it can occur 0 or more times, to give us any number of a's that is divisible by 3.

(b) Another way to think of this is that there are three optional a's and all the b's you want. That gives us:

$b^* (a \cup \varepsilon) b^* (a \cup \varepsilon) b^* (a \cup \varepsilon) b^*$

(c) Another way to think of this is that we need one instance of aaa. All other instances of aa must occur with

either b or end of string on both sides. The aaa can occur anywhere so we'll plunk it down, then list the options for everything else twice, once on each side of it:

$(ab \cup aab \cup b)^* \quad aaa \quad (ba \cup baa \cup b)^*$

4. (a) True. Consider the defining regular expression: $a^*b^*a^*b^*$. To get baa, take no a's, then one b, then two a's then no b's.

(b) True. We can prove that two sets X and Y are equal by showing that any string in X must also be in Y and vice versa. First we show that any string in $b^*a^* \cap a^*b^*$ (which we'll call X) must also be in $a^* \cup b^*$ (which we'll call Y). Any string in X must have two properties: (from b^*a^*): all b's come before all a's; and (from a^*b^*): all a's come before all b's. The only way to have both of these properties simultaneously is to be composed of only a's or only b's. That's exactly what it takes to be in Y.

Next we must show that every string in Y is in X. Every string in Y is either of the form a^* or b^*. All strings of the form a^* are in X since we simply take b^* to be b^0, which gives us $a^* \cap a^* = a^*$. Similarly for all strings of the form b^*, where we take a^* to be a^0.

(c) False. Remember that to show that any statements is false it is sufficient to find a single counterexample:

$\varepsilon \in a^*b^*$ and $\varepsilon \in c^*d^*$. Thus $\varepsilon \in a^*b^* \cap c^*d^*$, which is therefore not equal to ∅.

(d) False. There is no way to generate abcd from $(a(cd)^*b)^*$. Let's call the language generated by $(a(cd)^*b)^*$ L. Notice that every string in L has the property that every instance of $(cd)^*$ is immediately preceded by a. abcd does not possess that property.

5. That the language on the right is included in the language on the left is immediately apparent since every string in the right-hand language is a string of a's and b's. To show that any string of a's and b's is contained in the language on the right, we note that any such string begins with zero or more a's. If there are no b's, then the string is contained in a^*. If there is at least one b, we strip off any initial a's as a part of a^* and examine the remainder. If there are no more b's, the remainder is in ba^*. If there is at least one more b to the right, then we strip of the initial b and any following consecutive a's (a string in ba^*) and examine the remainder. Repeat the last two steps until the end of the string is reached. Thus, every string of a's and b's is included in the language on the right.

6. (i) a, c, e (b contains superscript n; d contains ∩)

(ii) (a) = $(a \cup b)^*$

(c) = ∅

(e) = b^*

(iii) a, c, d, e (b is $\{a^m b^n : m > n\}$, which is not regular)

7. (a) F, (b) F, (c) T, (d) F, (e) T, (f) T, (g) T, (h) F, (i) T, (j) T, (k) F, (l) T, (m) T, (n) F, (o) F, (p) T (by def. of +), (q) T, (r) F, (s) T, (t) F, (u) F, (v) T.

8. $(a \cup ba)^* (\varepsilon \cup b \cup bbb^*) = (a \cup ba)^*b^*$

9. (a) (1) no (2) no, (3) yes, (4) yes
L is composed of strings whose second half is the reverse of the first half.

(b) (1) no (2) no (3) no (4) yes
L contains only the empty string.

(c) (1) no (2) yes (3) no (4) yes
L contains strings of even length whose first half is the same as the second half. To see why this is so, notice that $|uu|$ is necessarily even, since it's $|u|$ times 2. So we must assure that $|www|$ is also even. This will only happen if $|w|$ is even. To discover what *u* is for any proposed *w*, we must first write out *www*. Then we split it in half and call that *u*. Suppose that *w* can be described as the concatenation of two strings of equal length, *r* and *s*. (We know we can do this, since we already determined that $|w|$ is even.) Then *w* will be equal to *rs* and *www* will be *rsrsrs*. So *u* must equal both *rsr* and *srs*. There can only be such a *u* if *r* and *s* are the same.

10. $(\varepsilon \cup ((1\text{-}9)(0\text{-}9)^*))(1\cup3\cup5\cup7\cup9)$, or, without using ε or the dash notation,

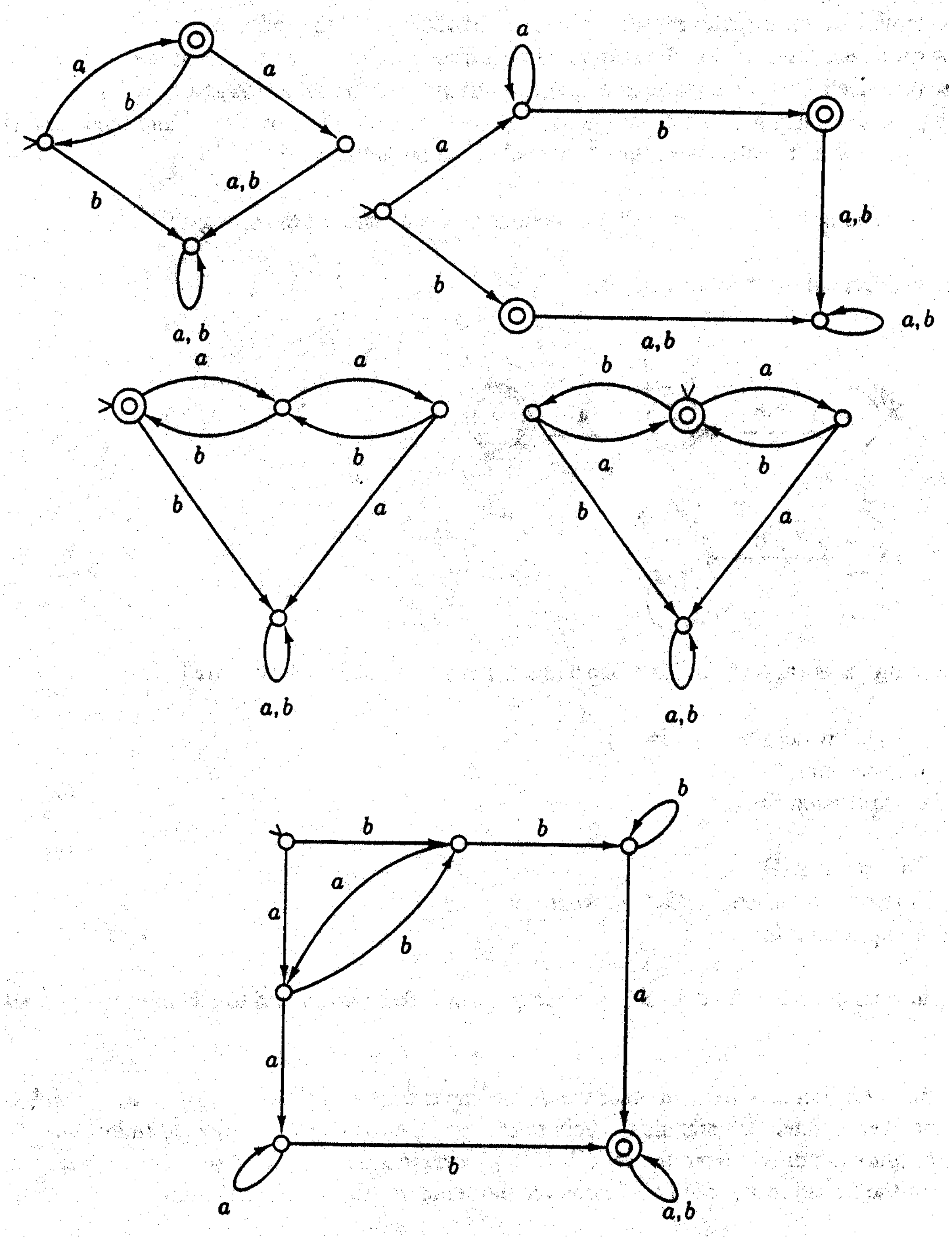
$$(1\cup3\cup5\cup7\cup9) \cup$$
$$((1\cup2\cup3\cup4\cup5\cup6\cup7\cup8\cup9)\ (0\cup1\cup2\cup3\cup4\cup5\cup6\cup7\cup8\cup9)^*\ (1\cup3\cup5\cup7\cup9))$$

11. Whew. A lot of formalism. The key is to walk through it one step at a time. It’s good practice. R relates pairs of strings that are identical except that the second one has one extra b concatenated on the end. So it includes, for example, {(a, ab), (ab, abb), (abb, abbb), (b, bb), (bb, bbb), …}. Now we have to compute R′. Consider the element a. First, we must add (a, a) to make R′ reflexive. Now we must consider transitivity. R gives us (a, ab). But it also gives us (ab, abb), so, by transitivity, R′ must contain (a, abb). In fact, a must be related to all strings in the language ab*. Similarly ε must be related to all strings in εb* or simply b*. And b must be related to all strings in bb*. We could also notice many other things, such as the fact that ab is related to all strings in abb*, but we don’t need to bother to do that to solve this problem. What we need to do is to figure out what L′ is. It’s all strings that are related via R′ to some element of L. There are three elements of L, {ε, a, b}. So L′ = b* ∪ ab* ∪ bb*. But every string in bb* is also in b*, so we can simplify to b* ∪ ab*.

CS 341 Homework 4
Deterministic Finite Automata

1. If M is a deterministic finite automaton. Under exactly what circumstances is ε ∈ L(M)?

2. Describe informally the languages accepted by each of the following deterministic FSMs:



(from Elements of the Theory of Computation, H. R. Lewis and C. H. Papdimitriou, Prentice-Hall, 1998.)

3. Construct a deterministic FSM to accept each of the following languages:

(a) {w ∈ {a, b}* : each ‘a’ in w is immediately preceded and followed by a ‘b’}

(b) {w ∈ {a, b}* : w has abab as a substring}

(c) {w ∈ {a, b}* : w has neither aa nor bb as a substring}

(d) {w ∈ {a, b}* : w has an odd number of a's and an even number of b's}

(e) {w ∈ {a, b}* : w has both ab and ba as substrings}

4. Construct a deterministic finite state transducer over {a, b} for each of the following tasks:

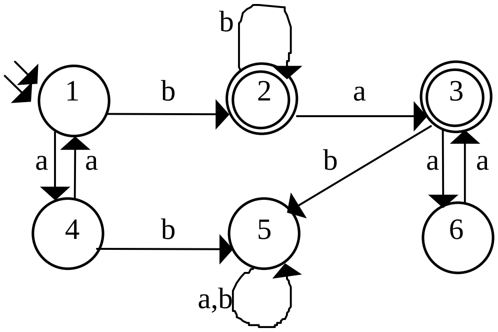
(a) On input w produce a^n, where n is the number of occurrences of the substring ab in w.

(b) On input w produce a^n, where n is the number of occurrences of the substring aba in w.

(c) On input w produce a string of length w whose i^{th} symbol is an a if i = 1 or if i > 1 and the i^{th} and $(i-1)^{st}$ symbols of w are different; otherwise, the i^{th} symbol of the output is b.

5. Construct a dfa accepting L = {w ∈ {a, b}* : w contains no occurrence of the string ab}.

6. What language is accepted by the following fsa?



7. Give a dfa accepting {x ∈ {a, b}* : at least one a in x is not immediately followed by b}.

8. Let L = {w ∈ {a, b}* : w does not end in ba}.

(a) Construct a dfa accepting L.

(b) Give a regular expression for L.

9. Consider L = {a^nb^n : 0 ≤ n ≤ 4}

(a) Show that L is regular by giving a dfa that accepts it.

(b) Give a regular expression for L.

10. Construct a deterministic finite state machine to accept strings that correspond to odd integers *without* leading zeros.

11. Imagine a traffic light. Let Σ = {a}. In other words, the input consists just of a string of a's. Think of each a as the output from a timer that signals the light to change. Construct a deterministic finite state transducer whose outputs are drawn from the set {Y, G, R} (corresponding to the colors yellow, green, and red). The outputs of the transducer should correspond to the standard traffic light behavior.

12. Recall the finite state machine that we constructed in class to accept $1.00 in change or bills. Modify the soda machine so that it actually does something (i.e., some soda comes out) by converting our finite state acceptor to a finite state transducer. Let there be two buttons, one for Coke at $.50 and one for Water at $.75 (yes, it's strange that water costs more than Coke. The world is a strange place). In any case, there will now be two new symbols in the input alphabet, C and W. The machine should behave as follows:

- The machine should keep track of how much money has been inserted. If it ever gets more than $1.50, it should spit back enough to get it under $1.00 but keep it above $.75.
- If the Coke or Water button is pushed and enough money has been inserted, the product and the change should be output.
- If a button is pushed and there is not enough money, the machine should remember the button push and wait until there is enough money, at which point it should output the product and the change.

13. Consider the problem of designing an annoying buzzer that goes off whenever you try to drive your car and you're not wearing a seat belt. (For simplicity, we'll just worry about the driver's possible death wish. If you want to make this harder, you can worry about the other seats as well.) Design a finite state transducer whose inputs are drawn from the alphabet {KI, KR, SO, SU, BF, BU}, representing the following events, respectively: "key just inserted into ignition", "key just removed from ignition", "seat just became occupied", "seat just became unoccupied", "belt has just been fastened", and "belt has just been unfastened". The output alphabet is {ON, OFF}. The buzzer should go on when ON is output and stay off until OFF is output.

14. Is it possible to construct a finite state transducer that can output the following sequence:

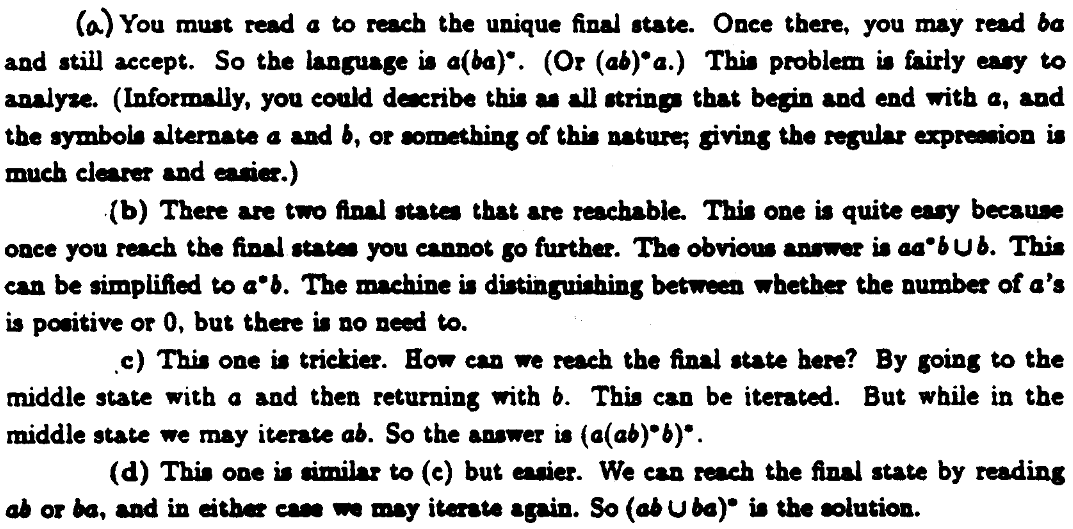
1010010001000010000010000001…

If it is possible, design one. If it's not possible, why not?

Solutions

1. $\varepsilon \in$ L(M) iff the initial state is a final state. Proof: M will halt in its initial state given ε as input. So: (IF) If the initial state is a final state, then when M halts in the initial state, it will be in a final state and will accept ε as an element of L(M). (ONLY IF) If the initial state is not a final state, then when M halts in the initial state, it will reject its input, namely ε. So the only way to accept ε is for the initial state to be a final state.

2.



(a) You must read a to reach the unique final state. Once there, you may read ba and still accept. So the language is $a(ba)^*$. (Or $(ab)^*a$.) This problem is fairly easy to analyze. (Informally, you could describe this as all strings that begin and end with a, and the symbols alternate a and b, or something of this nature; giving the regular expression is much clearer and easier.)

(b) There are two final states that are reachable. This one is quite easy because once you reach the final states you cannot go further. The obvious answer is $aa^*b \cup b$. This can be simplified to a^*b. The machine is distinguishing between whether the number of a's is positive or 0, but there is no need to.

c) This one is trickier. How can we reach the final state here? By going to the middle state with a and then returning with b. This can be iterated. But while in the middle state we may iterate ab. So the answer is $(a(ab)^*b)^*$.

(d) This one is similar to (c) but easier. We can reach the final state by reading ab or ba, and in either case we may iterate again. So $(ab \cup ba)^*$ is the solution.

(e) Number the states 1,2,3,4,5,6 going right to left, top to bottom. The following properties characterize each state:

1: e has been read.
2: xb has been read, for some $x \in (a \cup b)^*$ not ending in b.
3: xbb has been read, for some $x \in (a \cup b)^*$.
4: xa has been read, for some $x \in (a \cup b)^*$ not ending in a.
5: xaa has been read, for some $x \in (a \cup b)^*$.
6: $xbbay$ or $xaaby$ has been read, for some $x, y \in (a \cup b)^*$.

Therefore the language is all strings containing bba or aab as a substring, i.e., $(a \cup b)^*(bba \cup aab)(a \cup b)^*$.

3.

(a) $L = \{w \in \{a,b\}^* :$ each a in w is immediately preceded and immediately followed by a $b\}$.

(A regular expression for L is $(b^*ba)(b^*ba)^*bb^* \cup b^*$, or, using $^+$, $(b^+a)^+b^+ \cup b^*$. Notice the necessary distinction between strings with no a's and those with a's. Why doesn't the simpler $b^*(b^+ab^+)^*$ work?)

This will need a machine with a deadstate because as soon as we see an a not preceded or followed by a b, the string should be rejected and no matter what comes later, the string is bad. I.e., we will assume the string is ok until a specific occurence which tells us to reject the string.

Clearly $e \in L$ since every a in e has the property. Now for any longer string, the machine only needs to remember what the last symbol was to determine if the string should be rejected.

So we could make states with the properties:

1: $e \in L$ has been read.
2: xa has been read, for some $x \in L$ not ending in a (the string so far is ok, but we'd better see a b next since $xa \notin L$.)
3: $xb \in L$ has been read (the string so far is ok.)
4: x has been read, such that for no y is $xy \in L$. (we know the string is bad – no matter what comes later.)

You should be able to draw the machine now. Notice that $s = 1$, $F = \{1, 3\}$.

(b) $L = \{w \in \{a,b\}^* : w$ has $abab$ as a substring$\}$.

(A regular expression for L is easy: $(a \cup b)^*abab(a \cup b)^*$.)

Again we need to keep track only of the last part of the string, in this case the last 3 symbols. In this one we are looking for an occurence in the string which *will* make us accept the string (compare to Problem (a).) Once there has been an occurence of $abab$, whatever follows is irrelevant.

Here are the relevant properties of the string as it is read in:

1: x has been read, for some $x \in (a \cup b)^*$ such that $x \notin L$ and x does not end in a.
2: xa has been read, for some $x \in (a \cup b)^*$ such that $x \notin L$ and x does not end in ab.
3: xab has been read, for some $x \in (a \cup b)^*$ such that $x \notin L$ and x does not end in ab.
4: $xaba$ has been read, for some $x \in (a \cup b)^*$ such that $x \notin L$ and x does not end in ab.
5: $xababy$ has been read, for some $x, y \in (a \cup b)^*$ such that $x \notin L$ and x does not end in ab.

So a 5 state machine can do the trick. The start state is 1, because that's the property e has ($e \in (a \cup b)^*$ and e does not end in a.) Any string with property 1 which is then followed by b continues to have property 1, so $\delta(1, b) = 1$. Any string with property 1 which is then followed by a now has property 2, so $\delta(1, a) = 2$. And so on. Clearly $\delta(5, \sigma) = 5$ since once $abab$ has been seen, that fact cannot be changed – $abab$ continues to have been seen. Clearly a string has $abab$ as a substring iff it has property 5, so $F = \{5\}$. Now you draw the DFA.

(c) $L = \{w \in \{a, b\}^* : w$ has neither aa nor bb as a substring$\}$.

(A regular expression for L is $e \cup a(ba)^*(b \cup e) \cup b(ab)^*(a \cup e)$. This distinguishes between whether the string starts with a or b or is empty. Another one is $(a \cup e)(ba)^*(b \cup e)$, though this is perhaps less obvious.)

Like Problem (a), we should assume the string is ok until we see a bad occurence (aa or bb). To test this, we clearly only need to keep track of the last symbol read. So the relevant properties are:

1: e has been read (and so a or b may follow.)

2: xa has been read, for some $xa \in L$ (so only b may follow.)

3: xb has been read, for some $xb \in L$ (so only a may follow.)

4: x has been read, for some $x \notin L$.

Clearly any string with property 1, 2 or 3 is in L, so $F = \{1, 2, 3\}$. The start state is 1. Now you draw it.

(d) $L = \{w \in \{a, b\}^* : \#(a, w)$ is odd and $\#(b, w)$ is even$\}$.

I use the function $\#(\sigma, x)$ to mean "the number of occurences of symbol σ in string x." E.g., $\#(a, aba) = 2$ and $\#(b, aaa) = 0$.

Unlike the previous problems, there is no specific occurence we are looking for, either to reject or accept the string. Instead, we need to continually monitor it. When the string is all read in, its status will then determine whether it is accepted or rejected.

Clearly what we need to monitor is the parity (even or odd) of the number of a's and the number of b's. These are independent data, so there are $2 \times 2 = 4$ possible states or properties:

(0,0): x has been read, where $\#(a, x)$ and $\#(b, x)$ both even.

(0,1): x has been read, where $\#(a, x)$ even and $\#(b, x)$ odd.

(1,0): x has been read, where $\#(a, x)$ odd and $\#(b, x)$ even.

(1,1): x has been read, where $\#(a, x)$ and $\#(b, x)$ both odd.

Since $\#(\sigma, e) = 0$, and 0 is even, the start state is (0,0). (A fair number of people unnecessarily distinguish between 0 and other even numbers, producing machines with more states than necessary.) The only final state is (1,0). δ can be defined by $\delta((m, n), a) = (m + 1 \bmod 2, n)$ and $\delta((m, n), b) = (m, n + 1 \bmod 2)$.

This is a technique easily generalized. Finite automata cannot count to arbitrarily high natural numbers, but they can count modulo a number (so-called "clock arithmetic"). The DFA just given counts the number of a's and the number of b's modulo 2. (A number m is even iff m is congruent to 0 mod 2, written $x \equiv 0 \bmod 2$, e.g., $x = ..., -2, 0, 2, 4, ...$) You could design a DFA to accept all strings x with $\#(a, x) \equiv 7 \bmod 12$ and $\#(b, x) \equiv 0 \bmod 5$ and and $\#(c, x) \equiv 2 \bmod 3$, i.e., $\#(a, x) = 7, 19, 26, ...$ and $\#(b, x)$ is a multiple of 5 and

$\#(c, x) = 2, 5, 8, \ldots.$ A minimum state DFA to accept this language uses $12 \times 5 \times 3 = 180$ states. For notational convenience, I would call the states (i, j, k), where $0 \leq i \leq 12$, $0 \leq j \leq 5$ and $0 \leq k \leq 3$. Then the final state would be $(7, 0, 2)$. The start state is of course $(0, 0, 0)$.

What would you do if you wanted all strings x with $\#(a, x) \equiv 2$ or $3 \bmod 4$, or $\#(b, x) \equiv 1 \bmod 3$? (Hint: the states are constructed in the same manner; only the final conditions are different.)

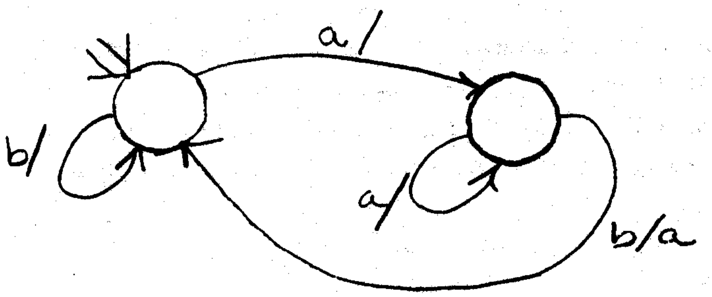
(e) $L = \{w \in \{a, b\}^* : w$ has both ab and ba as substrings$\}$.

Here we are looking for not one occurence but two in the string. There are two subtleties. Either event might occur first, so we must be prepared for the ab or the ba to be read first. Also, the definition of L does not require the two substrings of ab anb ba to be nonoverlapping: e.g., $aba \in L$.

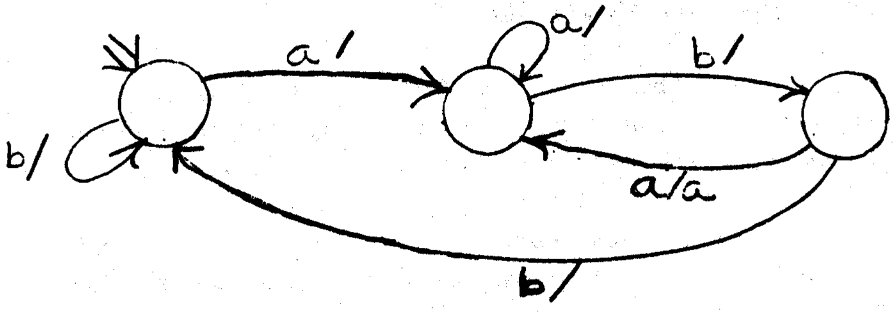
1: e has been read (so we have seen neither substring.)
2: a^m has been read, for some $m \geq 1$ (so we have seen neither substring.)
3: $a^m b^n$ has been read, for some $m, n \geq 1$ (so we have seen ab.)
4: b^m has been read, for some $m \geq 1$ (so we have seen neither substring.)
5: $b^m a^n$ has been read, for some $m, n \geq 1$ (so we have seen ba.)
6: $a^m b^n a x$ or $b^m a^n b x$ has been read, for some $m, n \geq 1$ and $x \in (a \cup b)^*$ (so we have seen ab and ba.)

Clearly, 1 is the start state and $\{6\}$ is the set of final states. You should be able to draw the DFA now.

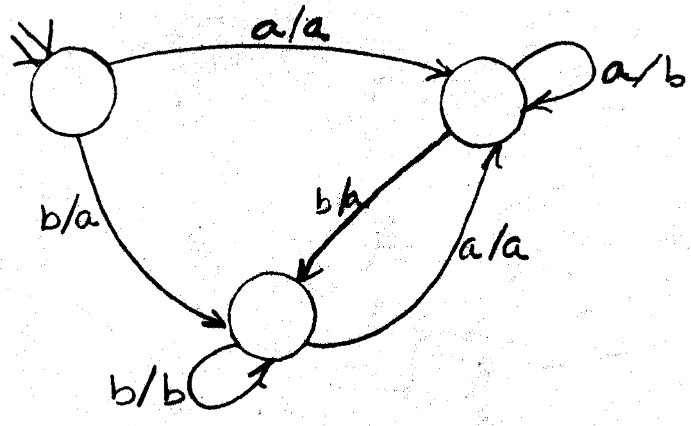
4. (a)



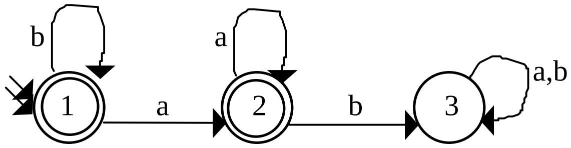
(b)



(c)

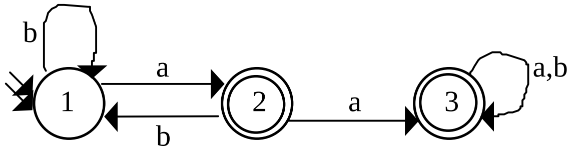


5.

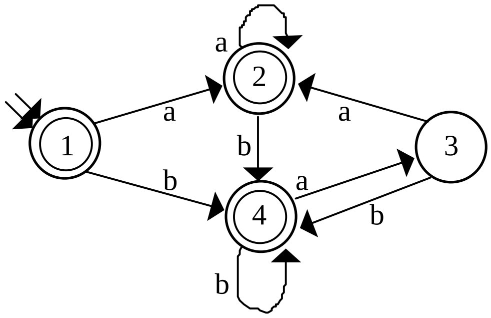


6. (aa)* (bb* ∪ bb*a(aa)*) = (aa)*b$^+$(ε ∪ a(aa)*) = all strings of a's and b's consisting of an even number of a's, followed by at least one b, followed by zero or an odd number of a's.

7.

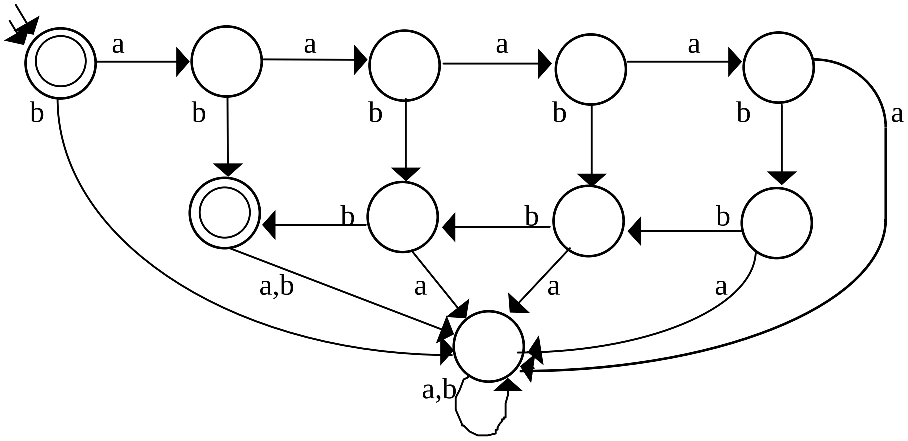


8. (a)



(b) ε ∪ a ∪ (a ∪ b)* (b ∪ aa)

9. (a)



(b) (ε ∪ ab ∪ aabb ∪aaabbb ∪aaaabbbb)

CS 341 Homework 5
Regular Expressions in UNIX

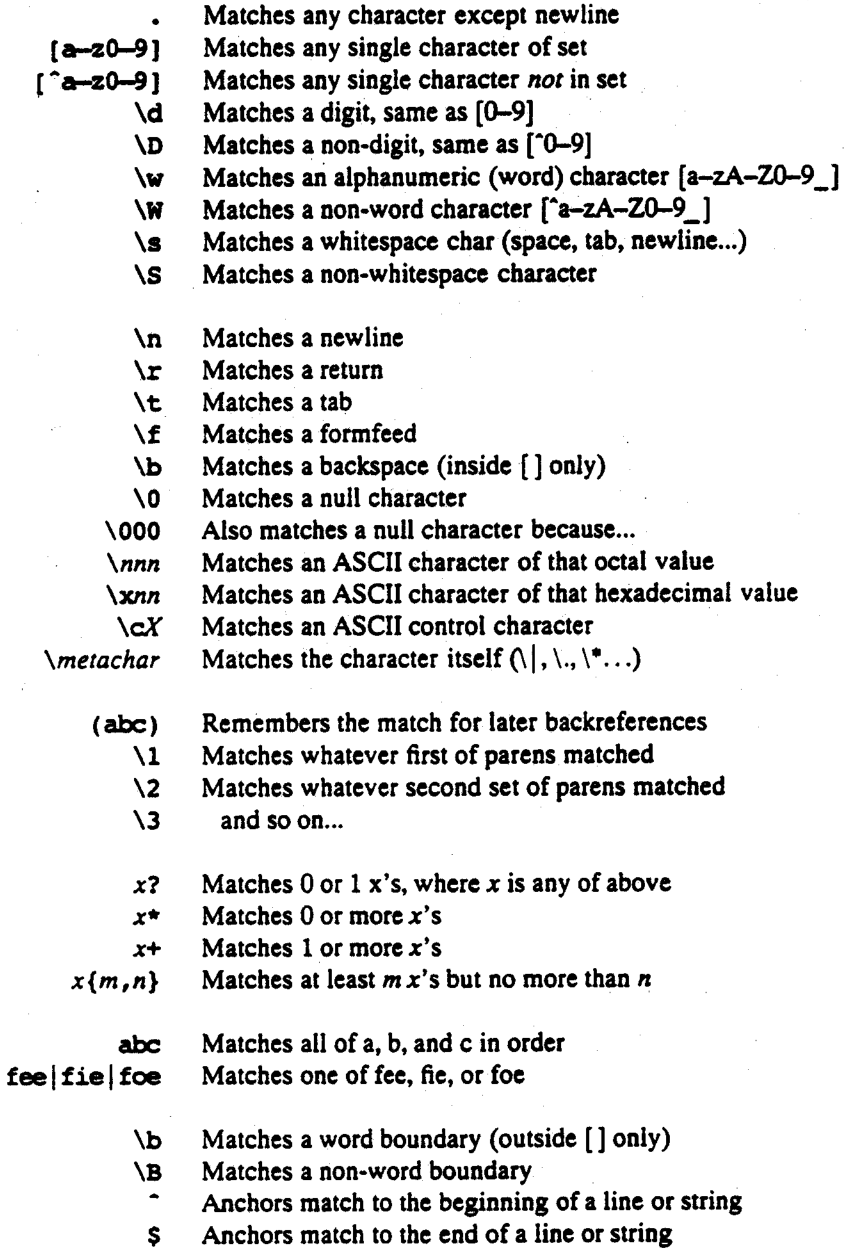
Regular expressions are all over the place in UNIX, including the programs grep, sed, and vi. There's a regular expression pattern matcher built into the programming language perl. There's also one built into the majordomo maillist program, to be used as a way to filter email messages. So it's easy to see that people have found regular expressions extremely useful. Each of the programs that uses the basic idea offers its own definition of what a regular expression is. Some of them are more powerful than others. The definition in perl is shown on the reverse of this page.

1. Write perl regular expressions to do the following things. If you have easy access to a perl interpreter, you might even want to run them.

(a) match occurrences of your phone number
(b) match occurrences of any phone number
(c) match occurrences of any phone number that occurs more than once in a string
(d) match occurrences of any email address that occurs more than once in a string
(e) match the Subject field of any mail message from yourself
(f) match any email messages where the address of the sender occurs in the body of the message

2. Examine the constructs in the perl regular expression definition closely. Compare them to the much more limited definition we are using. Some of them can easily be described in terms of the primitive capabilities we have. In other words, they don't offer additional power, just additional convenience. Some of them, though, are genuinely more powerful, in the sense that they enable you to define languages that aren't regular (i.e., they cannot be recognized with Finite State Machines). Which of the perl constructs actually add power to the system? What is it about them that makes them more powerful?

Regular Expressions in perl

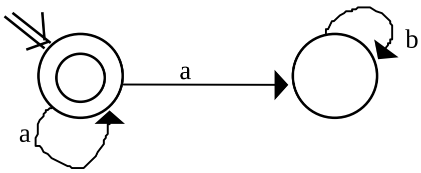


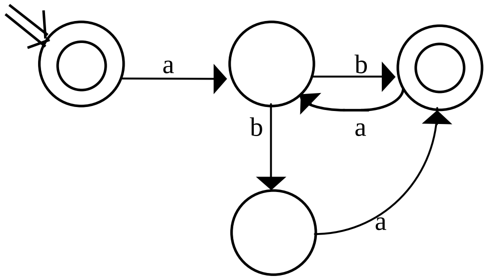
<table>
<tr><td>.</td><td>Matches any character except newline</td></tr>
<tr><td>[a–z0–9]</td><td>Matches any single character of set</td></tr>
<tr><td>[^a–z0–9]</td><td>Matches any single character <i>not</i> in set</td></tr>
<tr><td>\d</td><td>Matches a digit, same as [0–9]</td></tr>
<tr><td>\D</td><td>Matches a non-digit, same as [^0–9]</td></tr>
<tr><td>\w</td><td>Matches an alphanumeric (word) character [a–zA–Z0–9_]</td></tr>
<tr><td>\W</td><td>Matches a non-word character [^a–zA–Z0–9_]</td></tr>
<tr><td>\s</td><td>Matches a whitespace char (space, tab, newline...)</td></tr>
<tr><td>\S</td><td>Matches a non-whitespace character</td></tr>
<tr><td>\n</td><td>Matches a newline</td></tr>
<tr><td>\r</td><td>Matches a return</td></tr>
<tr><td>\t</td><td>Matches a tab</td></tr>
<tr><td>\f</td><td>Matches a formfeed</td></tr>
<tr><td>\b</td><td>Matches a backspace (inside [ ] only)</td></tr>
<tr><td>\0</td><td>Matches a null character</td></tr>
<tr><td>\000</td><td>Also matches a null character because...</td></tr>
<tr><td>\<i>nnn</i></td><td>Matches an ASCII character of that octal value</td></tr>
<tr><td>\x<i>nn</i></td><td>Matches an ASCII character of that hexadecimal value</td></tr>
<tr><td>\c<i>X</i></td><td>Matches an ASCII control character</td></tr>
<tr><td>\<i>metachar</i></td><td>Matches the character itself (\|, \., *...)</td></tr>
<tr><td>(abc)</td><td>Remembers the match for later backreferences</td></tr>
<tr><td>\1</td><td>Matches whatever first of parens matched</td></tr>
<tr><td>\2</td><td>Matches whatever second set of parens matched</td></tr>
<tr><td>\3</td><td>and so on...</td></tr>
<tr><td><i>x</i>?</td><td>Matches 0 or 1 x's, where <i>x</i> is any of above</td></tr>
<tr><td><i>x</i>*</td><td>Matches 0 or more <i>x</i>'s</td></tr>
<tr><td><i>x</i>+</td><td>Matches 1 or more <i>x</i>'s</td></tr>
<tr><td><i>x</i>{<i>m</i>,<i>n</i>}</td><td>Matches at least <i>m</i> <i>x</i>'s but no more than <i>n</i></td></tr>
<tr><td>abc</td><td>Matches all of a, b, and c in order</td></tr>
<tr><td>fee|fie|foe</td><td>Matches one of fee, fie, or foe</td></tr>
<tr><td>\b</td><td>Matches a word boundary (outside [ ] only)</td></tr>
<tr><td>\B</td><td>Matches a non-word boundary</td></tr>
<tr><td>^</td><td>Anchors match to the beginning of a line or string</td></tr>
<tr><td>$</td><td>Anchors match to the end of a line or string</td></tr>
</table>

from Programming in Perl, Larry Wall and Randall L. Scwartz, O'Reilly & Associates, 1990.

CS 341 Homework 6
Nondeterministic Finite Automata

1. (a) Which of the following strings are accepted by the nondeterministic finite automaton shown on the left below?
(i) a
(ii) aa
(iii) aab
(iv) ε





(b) Which of the following strings are accepted by the nondeterministic finite automaton on the right above?
(i) ε
(ii) ab
(iii) abab
(iv) aba
(v) abaa

2. Write regular expressions for the languages accepted by the nondeterministic finite automata of problem 1.

3. For any FSM F, let |F| be the number of states in F. Let R be the machine shown on the right in problem 1.
Let L = {w ∈ {0, 1}* : ∃M such that M is an FSM, L(M) = L(R), |M| ≥ |R|, and w is the binary encoding of |M|}. Write a regular expression for L.

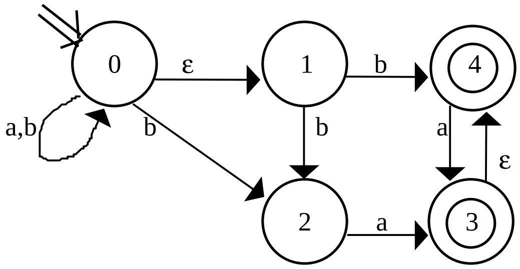
4. Draw state diagrams for nondeterministic finite automata that accept these languages:
(a) (ab)*(ba)* ∪ aa*
(b) ((ab ∪ aab)*a*)*
(c) ((a*b*a*)*b)*
(d) (ba ∪ b)* ∪ (bb ∪ a)*

5. Some authors define a nondeterministic finite automaton to be a quintuple (K, Σ, Δ, S, F), where K, Σ, Δ, and F are as we have defined them and S is a finite set of initial states, in the same way that F is a finite set of final states. The automaton may nondeterministically begin operating in any of these initial states. Explain why this definition is not more general than ours in any significant way.

6. (a) Find a simple nondeterministic finite automaton accepting ((a ∪ b)*aabab).
(b) Convert the nondeterministic finite automaton of Part (a) into a deterministic finite automaton by the method described in class and in the notes.
(c) Try to understand how the machine constructed in Part (b) operates. Can you find an equivalent deterministic machine with fewer states?

7. Construct a NDFSA that accepts the language (ba ∪ ((a ∪ bb) a*b)).

8. Construct a deterministic finite automaton equivalent to the following nondeterministic automaton:



9. L = { w ∈ {a, b}* : every a is followed by at least one b }

(a) Write a regular expression that describes L.
(b) Write a regular grammar that describes L.
(c) Construct an FSM that accepts precisely L.

10. Consider the following regular grammar, which defines a language L:

S -> bF
S -> aS
F -> ε
F -> bF
F -> aF

(a) Construct an FSM that accepts precisely L.
(b) Write a regular expression that describes L.
(c) Describe L in English.

Solutions

1. (a) [i.] yes [ii.] yes [iii.] no [iv.] yes
(b) [i.] yes [ii.] yes [iii.] yes [iv.] yes [v.] no

2. (a) a* Note that the second state could be eliminated, since there's no path from it to a final state.
(b) (ab ∪ aba)* Notice that we could eliminate the start state and make the remaining final state the start state and we'd still get the same result.

3. To determine L, we need first to consider the set of machines that accept the same language as R. It turns out that we don't actually need to know what all such machines look like because we can immediately see that there's at least one with four states (R), one with 5, one with 6, and so forth, and all that we need to establish L is the sizes of the machines, not their structures.. From R, we can construct an infinite number of equivalent machines by adding any number of redundant states. For example, we could add a new, nonfinal state 1 that is reachable from the start state via an ε transition. Since 1 is not final and it doesn't go anywhere, it cannot lead to an accepting path, so adding it to R has no affect on R's behavior. Now we have an equivalent machine with 5 states. We can do it again to yield 6, and so forth. Thus the set of numbers we need to represent is simply $4 \leq n$. Now all we have to do is to describe the binary encodings of these numbers. If we want to disallow leading zeros, we get 1(0 ∪ 1) (0 ∪ 1) (0 ∪ 1)*. There must be at least three digits of which the first must be 1.

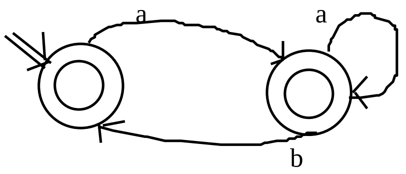
4. (a) The easiest way to do this is to make a 2 state FSA for aa* and a 4 state one for (ab)*(ba)*, then make a seventh state, the start state, that nondeterministically guesses which class an input string will fall into.
(b) First we simplify. ((ab ∪ aab)*a*)*

/ $(L_1^*L_2^*)^* = (L_1 \cup L_2)^*$ /

((ab ∪ aab) ∪ a)*

/ union is associative /

(ab ∪ aab ∪ a)*, which can be rewritten as (ab ∪ a)*. This is so because aab can be formed by one application a, followed by one of ab. So it is redundant inside a Kleene star. Now we can write a two state machine:



If you put the loop on a on the start state, either in place of where we have it, or in addition to it, it's also right.

(c) First we simplify: ((a*b*a*)*b)*

/ $(L_1{}^*L_2{}^*L_3{}^*)^* = (L_1 \cup L_2 \cup L_3)^*$ /

((a ∪ b ∪ a)*b)*

/ union is idempotent /

((a ∪ b)*b)*

There is a simple 2 state NDFSM accepting this, which is the empty string and all strings ending with b.

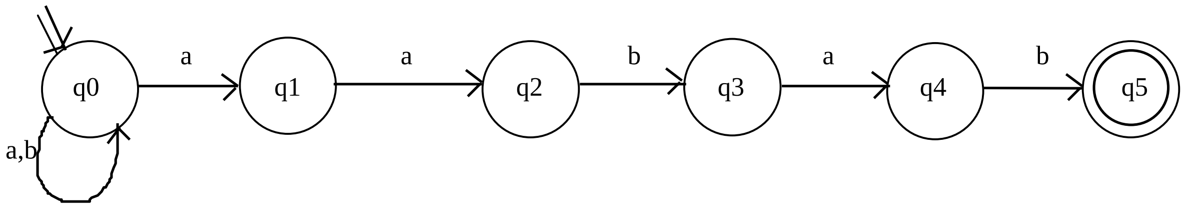
(d) This is the set of strings where either: (1) every a is preceded by a b,
or (2) all b's occur in pairs.

So we can make a five state nondeterministic machine by making separate machines (each with two states) for the two languages and then introducing ε transitions from the start state to both of them.

5. To explain that any construct A is not more general or powerful than some other construct B, it suffices to show that any instance of A can be simulated by a corresponding instance of B. So in this case, we have to show how to take a multiple start state NDFSA, A, and convert it to a NDFSA, B, with a single start state. We do this by initially making B equal to A. Then add to B a new state we'll call S0. Make it the only start state in B. Now add ε transitions from S0 to each of the states that was a start state in A. So B has a single start state (thus it satisfies our original definition of a NDFSA), but it simulates the behavior of A since the first thing it does is to move, nondeterministically, to all of A's start states and then it exactly mimics the behavior of A.

6. If you take the state machine as it is given, add a new start state and make ε transitions from it to the given start states, you have an equivalent machine in the form that we’ve been using.

7. (a)



(b) (1) Compute the E(q)s. Since there are no ε transitions, E(q), for all states q is just {q}.

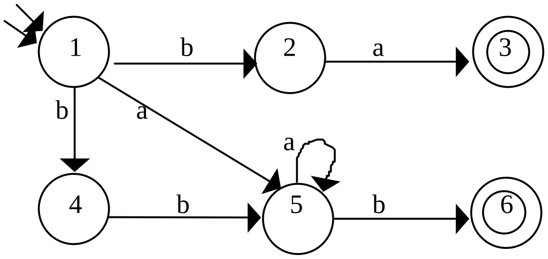
(2) S’ = {q0}

(3) δ’ = {
({q0}, a, {q0, q1}),
({q0}, b, {{q0}),
({q0, q1}, a, {q0, q1, q2}),
({q0, q1}, b, {q0}),
({q0, q1, q2}, a, {q0, q1, q2}),
({q0, q1, q2}, b, {q0, q3}),
({q0, q3}, a, {q0, q1, q4}),
({q0, q3}, b, {q0}),
({q0, q1, q4}, a, {q0, q1, q2}),
({q0, q1, q4}, b, {q0, q5}),
({q0, q5}, a, {q0, q1}),
({q0, q5}, b, {q0}) }

(4) K’ = {{q0}, {q0, q1}, {q0, q1, q2}, {q0, q3}, {q0, q1, q4}, {q0, q5}}
(5) F’ = {{q0, q5}}

(c) There isn’t a simpler machine since we need a minimum of six states in order to keep track of how many characters (between 0 and 5) of the required trailing string we have seen so far.

8. We can build the following machine really easily. We make the path from 1 to 2 to 3 for the ba option. The rest is for the second choice. We get a nondeterministic machine, as we generally do when we use the simple approach.

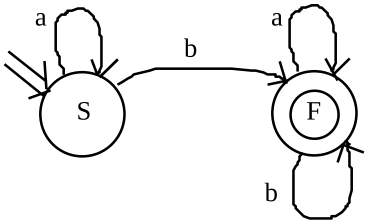


In this case, we could simplify our machine if we wanted to and get rid of state 4 by adding a transition on b from 2 to 5.

9. (1) E(q0) = {q0, q1}, E(q1) = {q1}, E(q2) = {q2}, E(q3) = {q3, q4}, E(q4) = {q4}
(2) s’ = {q0, q1}
(3) δ’ = { ({q0, q1}, a, {q0, q1}),
({q0, q1}, b, {q0, q1, q2, q4}),
({q0, q1, q2, q4}, a, {q0, q1, q3, q4}),
({q0, q1, q2, q4), b, {q0, q1, q2, q4}) }
({q0, q1, q3, q4}, a, {q0, q1, q3, q4}),
({q0, q1, q3, q4}, b, {q0, q1, q2, q4}),
(4) K’ = { {q0, q1}, {q0, q1, q3, q4}, {q0, q1, q2, q4} }
(5) F’ = { {q0, q1, q3, q4}, {q0, q1, q2, q4} }

This machine corresponds to the regular expression a*b(a ∪ b)*

10. (a)



(b) (a ∪ b)*ba* OR a*b(a ∪ b)*

(c) L = { w ∈ {a, b}* : there is at least one b }

CS 341 Homework 7
Review of Equivalence Relations

1. Assume a finite domain that includes just the specific cities mentioned here. Let R = the reflexive, symmetric, transitive closure of:

(Austin, Dallas), (Dallas, Houston), (Dallas, Amarillo), (Austin, San Marcos),
(Philadelphia, Pittsburgh), (Philadelphia, Paoli), (Paoli, Scranton),
(San Francisco, Los Angeles), (Los Angeles, Long Beach), (Long Beach, Carmel)

(a) Draw R as a graph.
(b) List the elements of the partition defined by R on its domain.

2. Let R be a relation on the set of positive integers. Define R as follows:

{(a, b) : (a mod 2) = (b mod 2)} In other words, R(a, b) iff a and b have the same remainder when divided by 2.

(a) Consider the following example integers: 1, 2, 3, 4, 5, 6. Draw the subset of R involving just these values as a graph.
(b) How many elements are there in the partition that R defines on the positive integers?
(c) List the elements of that partition and show some example elements.

3. Consider the language L, over the alphabet $\Sigma = \{a, b\}$, defined by the regular expression

$a^*(b \cup \varepsilon)\, a^*$

Let R be a relation on Σ^*, defined as follows:

R(x, y) iff both x and y are in L or neither x nor y is in L. In other words, R(x,y) if x and y have identical status with respect to L.

(a) Consider the following example elements of Σ^*: ε, b, aa, bb, aabaaa, bab, bbaabb. Draw the subset of R involving just these values as a graph.
(b) How many elements are there in the partition that R defines on Σ^*?
(c) List the elements of that partition and show some example elements.

Solutions

1. **(b)** [cities in Texas], [cities in Pennsylvania], [cities in California]

2. **(b)** Two
(c) [even integers] Examples: 2, 4, 6, 106
[odd integers] Examples: 1, 3, 5, 17, 11679

3. **(a)** (Hint: L is the language of strings with no more than one b.)
(b) Two
(c) [strings in L] Examples: ε, aa, b, aabaaa
[strings not in L] Examples: bb, bbaabb, bab

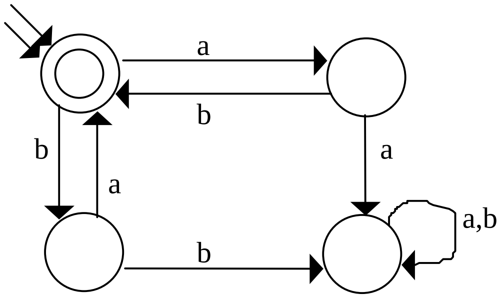
CS 341 Homework 8
Finite Automata, Regular Expressions, and Regular Grammars

1. We showed that the set of finite state machines is closed under complement. To do that, we presented a technique for converting a *deterministic* machine M into a machine M' such that L(M') is the complement of L(M). Why did we insist that M be deterministic? What happens if we interchange the final and nonfinal states of a nondeterministic finite automaton?

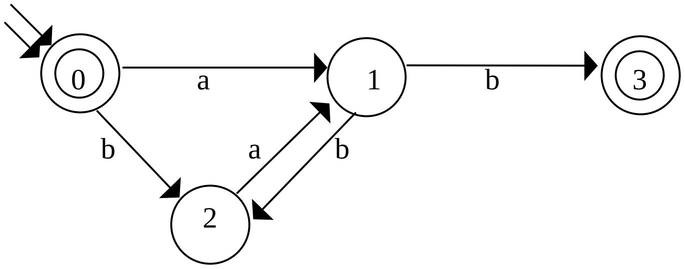
2. Give a direct construction for the closure under intersection of the languages accepted by finite automata. (Hint: Consider an automaton whose set of states is the Cartesian product of the sets of states of the two original automata.) Which of the two constructions, the one given in the text or the one suggested in this problem, is more efficient when the two languages are given in terms of nondeterministic finite automata?

3. Using the either of the construction techniques that we discussed, construct a finite automaton that accepts the language defined by the regular expression: a*(ab ∪ ba ∪ ε)b*.

4. Write a regular expression for the language recognized by the following FSM:



5. Consider the following FSM M:

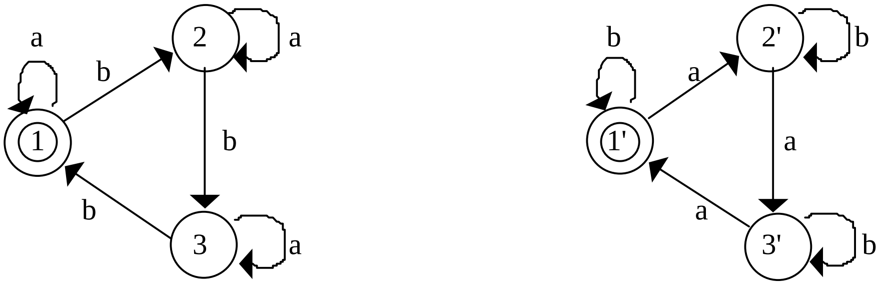


(a) Write a regular expression for the language accepted by M.
(b) Give a deterministic FSM that accepts the complement of the language accepted by M.

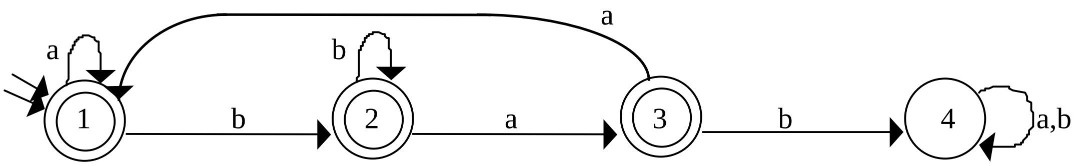
6. Construct a deterministic FSM to accept each of the following languages:
(a) (aba ∪ aabaa)*
(b) (ab)*(aab)*

7. Consider the language L = {w ∈ (a, b)* : w has an odd number of a's}
(a) Write a regular grammar for L.
(b) Use that grammar to derive a (possibly nondeterministic) FSA to accept L.

8. Construct a deterministic FSM to accept the intersection of the languages accepted by the following FSMs:



9. Consider the following FSM M:

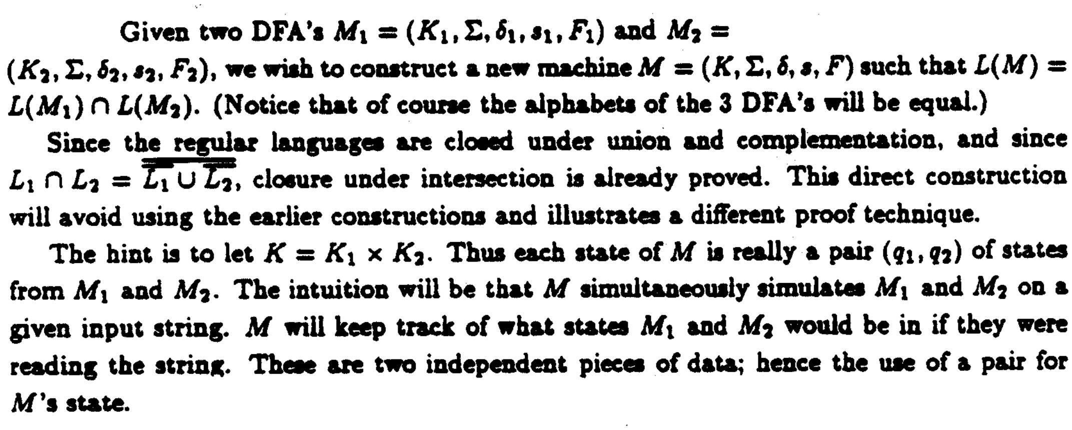


(a) Give a regular expression forL(M).
(b) Describe L(M) in English.

Solutions

1. We define acceptance for a NDFSA corresponding to the language L as there existing at least one path that gets us to a final state. There can be many other paths that don't, but we ignore them. So, for example, we might accept a string S that gets us to three different states, one of which accepts (which is why we accept the string) and two of which don't (but we don't care). If we simply flip accepting and nonaccepting states to get a machine that represents the complement of L, then we still have to follow all possible paths, so that same string S will get us to one nonaccepting state (the old accepting state), and two accepting states (the two states that previously were nonaccepting but we ignored). Unfortunately, we could ignore the superfluous nonaccepting paths in the machine for L, but now that those same paths have gotten us to accepting states, we can't ignore them, and we'll have to accept S. In other words, we'll accept S as being in the complement of L, even though we also accepted it as being in L. The key is that in a deterministic FSA, a rejecting path actually means reject. Thus it makes sense to flip it and accept if we want the complement of L. In a NDFSA, a rejecting path doesn't actually mean reject. So it doesn't make sense to flip it to an accepting state to accept the complement of L.

2.



Given two DFA's $M_1 = (K_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (K_2, \Sigma, \delta_2, s_2, F_2)$, we wish to construct a new machine $M = (K, \Sigma, \delta, s, F)$ such that $L(M) = L(M_1) \cap L(M_2)$. (Notice that of course the alphabets of the 3 DFA's will be equal.)

Since the regular languages are closed under union and complementation, and since $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, closure under intersection is already proved. This direct construction will avoid using the earlier constructions and illustrates a different proof technique.

The hint is to let $K = K_1 \times K_2$. Thus each state of M is really a pair (q_1, q_2) of states from M_1 and M_2. The intuition will be that M simultaneously simulates M_1 and M_2 on a given input string. M will keep track of what states M_1 and M_2 would be in if they were reading the string. These are two independent pieces of data; hence the use of a pair for M's state.

Initially, M_1 and M_2 start in their start states, s_1 and s_2. Therefore we should let $s = (s_1, s_2)$.

Now suppose that M_1 is in some state $q_1 \in K_1$ and reads symbol σ. What state does M_1 enter? $\delta_1(q_1, \sigma)$. Similarly for M_2. So we would like M, when in state (q_1, q_2) and reading σ, to enter state $(\delta_1(q_1,\sigma), \delta_2(q_2,\sigma))$; otherwise M would not be correctly keeping track of what M_1 and M_2 would do. So we define, for all $(q_1, q_2) \in K$ and all $\sigma \in \Sigma$,

$$\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)).$$

Notice that $\delta : K \times \Sigma \rightarrow K$, so everything is consistent and correct. Since $K = K_1 \times K_2$, this means δ is actually a function taking a pair of states (from M_1 and M_2) and a symbol from Σ.

We've now got the transitions defined, and M correctly simulates M_1 and M_2. I.e.,

$$\delta(s, x) = (q_1, q_2)$$

iff

$$\delta_1(s_1, x) = q_1 \text{ and } \delta_2(s_2, x) = q_2.[1]$$

So we only need to define F. When should M accept x? Exactly when both M_1 and M_2 do, since $x \in L(M_1) \cap L(M_2)$ iff $x \in L(M_1)$ and $x \in L(M_2)$. Therefore F should consist of all those states $(q_1, q_2) \in K$ such that $q_1 \in F_1$ and $q_2 \in F_2$. This can be written as

$$F = \{(q_1, q_2) : q_1 \in F_1 \text{ and } q_2 \in F_2\},$$

or more succinctly as $F = F_1 \times F_2$.

Thus the complete answer is

$$M = (K_1 \times K_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2)$$

where

$$\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)).$$

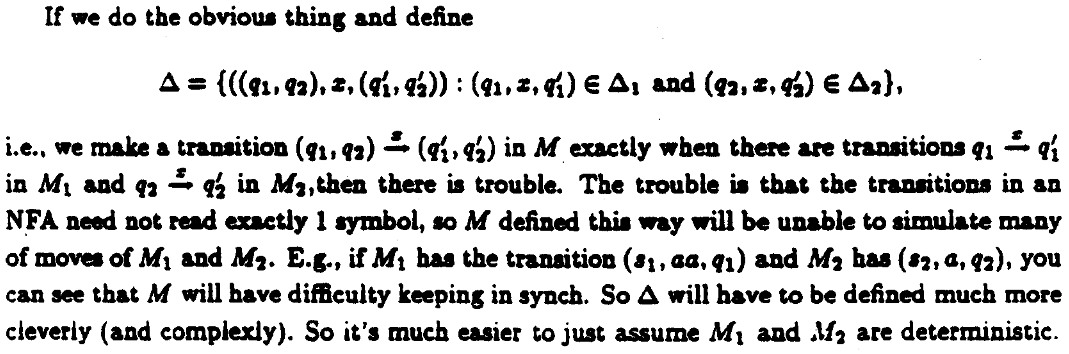
Notice that this assumes M_1 and M_2 are deterministic. What if M_1 and M_2 are not deterministic? We can assume that they are deterministic without loss of generality, because if they were not, the subset construction can be applied to them to produce equivalent DFA's. However, this construction can be modified to work directly on NFA's if desired. Unfortunately, it gets rather messy because of the following problem:

We are given two NFA's $M_1 = (K_1, \Sigma, \Delta_1, s_1, F_1)$ and $M_2 = (K_2, \Sigma, \Delta_2, s_2, F_2)$, and we wish to construct a new machine $M = (K, \Sigma, \Delta, s, F)$ such that $L(M) = L(M_1) \cap L(M_2)$.

[1] Technically, δ is a function of symbols not strings; however we can easily extend it to strings by the recursive generalization:

$$\begin{aligned} \delta(q, e) &= q \\ \delta(q, \sigma x) &= \delta(\delta(q, \sigma), x) \end{aligned}$$

I.e., if it is determined what δ does with a single symbol, then it is determined what δ does with a string simply by tracing through symbol by symbol.



If we do the obvious thing and define

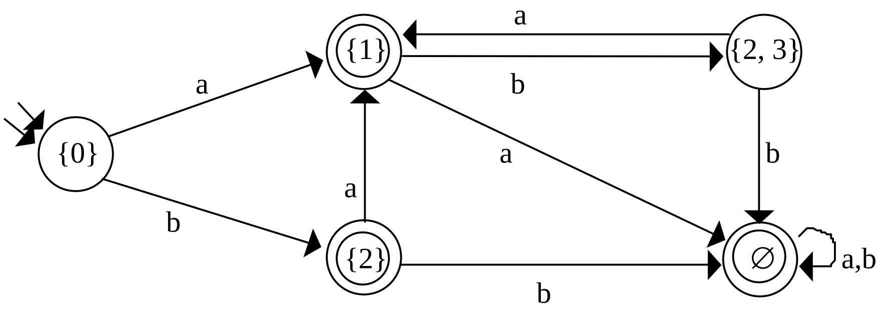
$$\Delta = \{((q_1, q_2), x, (q_1', q_2')) : (q_1, x, q_1') \in \Delta_1 \text{ and } (q_2, x, q_2') \in \Delta_2\},$$

i.e., we make a transition $(q_1, q_2) \xrightarrow{x} (q_1', q_2')$ in M exactly when there are transitions $q_1 \xrightarrow{x} q_1'$ in M_1 and $q_2 \xrightarrow{x} q_2'$ in M_2,then there is trouble. The trouble is that the transitions in an NFA need not read exactly 1 symbol, so M defined this way will be unable to simulate many of moves of M_1 and M_2. E.g., if M_1 has the transition (s_1, aa, q_1) and M_2 has (s_2, a, q_2), you can see that M will have difficulty keeping in synch. So Δ will have to be defined much more cleverly (and complexly). So it's much easier to just assume M_1 and M_2 are deterministic.

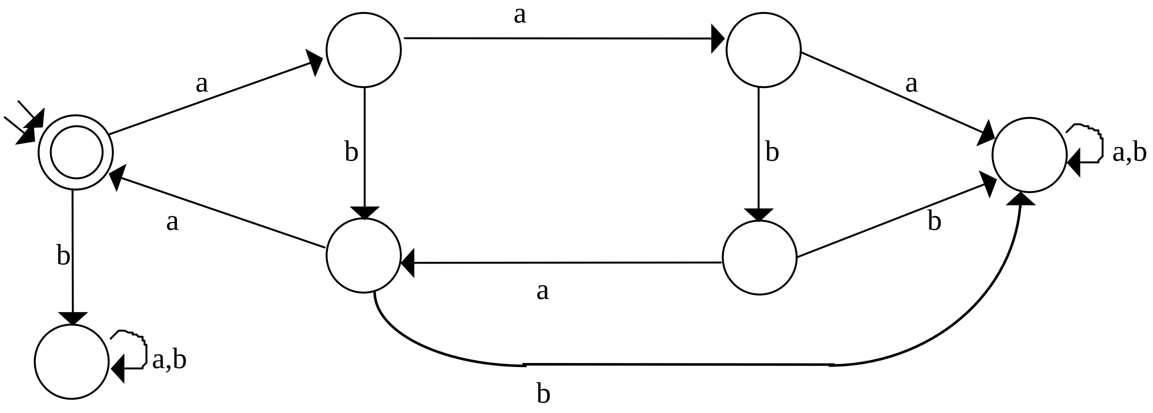
4. Without using the algorithm for finding a regular expression from an FSM, we can note in this case that the lower right state is a dead state, i.e., an absorbing, non-accepting state. We can leave and return to the initial state, the only accepting state, by reading ab along the upper path or by reading ba along the lower path. These can be read any number of times, in any order, so the regular expression is (ab ∪ ba)*. Note that ε is included, as it should be.

5. (a) ε ∪ ((a ∪ ba)(ba)*b)

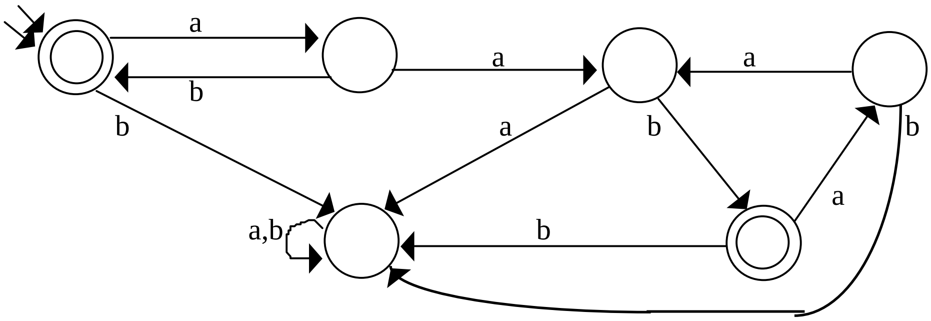
(b)



6. (a)



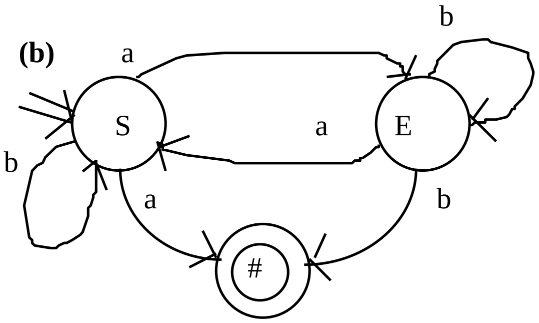
(b)



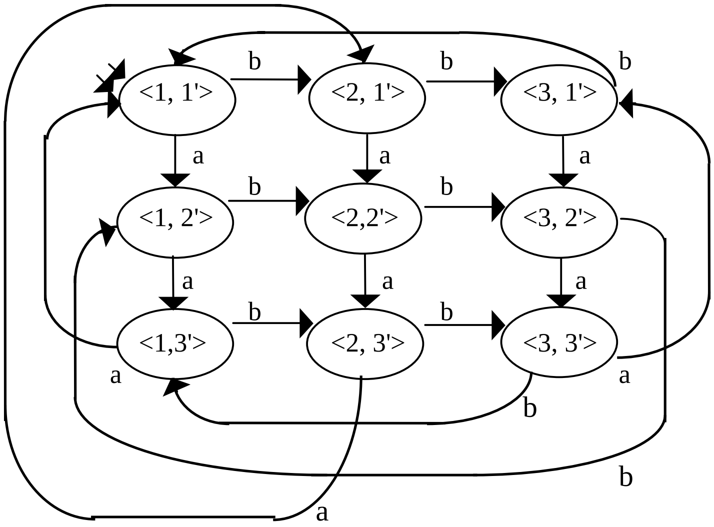
7. (a) Nonterminal S is the starting symbol. We'll use it to generate an odd number of a's. We'll also use the nonterminal E, and it will always generate an even number of a's. So, whenever we generate an a, we must either stop then, or we must generate the nonterminal E to reflect the fact that if we generate any more a's, we must generate an even number of them.

S → a
S → aE
S → bS
E → b
E → bE
E → aS

(b)



8.



9. (a) (a ∪ bb*aa)* (ε ∪ bb*(a ∪ ε))

(b) All strings in {a, b}* that contain no occurrence of bab.

CS 341 Homework 9
Languages That Are and Are Not Regular

1. Show that the following are not regular.
(a) $L = \{ww^R : w \in \{a, b\}^*\}$
(b) $L = \{ww : w \in \{a, b\}^*\}$
(c) $L = \{ww' : w \in \{a, b\}^*\}$, where w' stands for w with each occurrence of a replaced by b, and vice versa.

2. Show that each of the following is or is not a regular language. The decimal notation for a number is the number written in the usual way, as a string over the alphabet {-, 0, 1, …, 9}. For example, the decimal notation for 13 is a string of length 2. In unary notation, only the symbol 1 is used; thus 5 would be represented as 11111 in unary notation.
(a) L = {w : w is the unary notation for a natural number that is a multiple of 7}
(b) L = {w : w is the decimal notation for a natural number that is a multiple of 7}
(c) L = {w : w is the unary notation for a natural number n such that there exists a pair p and q of twin primes, both > n.} Two numbers p and q are a pair of twin primes iff q = p + 2 and both p and q are prime. For example, (3, 5) is a pair of twin primes.
(d) L = {w : w is, for some $n \geq 1$, the unary notation for 10^n}
(e) L = {w : w is, for some $n \geq 1$, the decimal notation for 10^n}
(f) L = {w is of the form x#y, where $x, y \in \{1\}^+$ and y = x+1 when x and y are interpreted as unary numbers} (For example, 11#111 and 1111#11111 ∈ L, while 11#11, 1#111, and 1111 ∉ L.)
(g) $L = \{a^n b^j : |n - j| = 2\}$
(h) $L = \{uww^R v : u, v, w \in \{a, b\}^+\}$
(i) $L = \{w \in \{a, b\}^* :$ for each prefix x of w, $\#a(x) \geq \#b(x)\}$

3. Are the following statements true or false? Explain your answer in each case. (In each case, a fixed alphabet Σ is assumed.)
(a) Every subset of a regular language is regular.
(b) Let $L' = L1 \cap L2$. If L' is regular and L2 is regular, L1 must be regular.
(c) If L is regular, then so is $L' = \{xy : x \in L \text{ and } y \notin L\}$.
(d) $\{w : w = w^R\}$ is regular.
(e) If L is a regular language, then so is $L' = \{w : w \in L \text{ and } w^R \in L\}$.
(f) If C is any set of regular languages, $\cup C$ (the union of all the elements of C) is a regular language.
(g) $L = \{xyx^R : x, y \in \Sigma^*\}$ is regular.
(h) If $L' = L1 \cup L2$ is a regular language and L1 is a regular language, then L2 is a regular language.
(i) Every regular language has a regular proper subset.
(j) If L1 and L2 are nonregular languages, then $L1 \cup L2$ is also not regular.

4. Show that the language $L = \{a^n b^m : n \neq m\}$ is not regular.

5. Prove or disprove the following statement:

If L_1 and L_2 are not regular languages, then $L_1 \cup L_2$ is not regular.

6. Show that the language $L = \{x \in \{a, b\}^* : x = a^n b a^m b a^{\max(m,n)}\}$ is not regular.

7. Show that the language $L = \{x \in \{a, b\}^* :$ x contains exactly two more b's than a's$\}$ is not regular.

8. Show that the language $L = \{x \in \{a, b\}^* :$ x contains twice as many a's as b's$\}$ is not regular.

9. Let L = {w : #a(w) = #b(w)}. (#a(w) = the number of a's in w.)
(a) Is L regular?
(b) Is L* regular?

Solutions

1. (a) L = {ww^R : w ∈ {a, b}*}. L is the set of all strings whose first half is equal to the reverse of the second half. All strings in L must have even length. If L is regular, then the pumping lemma tells us that ∃ N ≥ 1, such that ∀ strings w ∈ L, where |w| ≥ N, ∃ x, y, z, such that w = xyz, |xy| ≤ N, y ≠ ε, and ∀ q ≥ 0, xy^qz is in L. We must pick a string w ∈ L and show that it does not meet these requirements.

First, don't get confused by the fact that we must pick a string w, yet we are looking for strings of the form ww^R. These are two independent uses of the variable name w. It just happens that the problem statement uses the same variable name that the pumping lemma does. If it helps, restate the problem as L = {ss^R : s ∈ {a, b}*}.

We need to choose a "long" w, i.e., one whose length is greater than N. But it may be easier if we choose one that is even longer than that. Remember that the fact that |xy| ≤ N guarantees that y (the pumpable region) must occur within the first N characters of w. If we don't want to have to consider a lot of different possibilities for what y could be, it will help to choose a w with a long first region. Let's let w = a^Nbba^N. We know that y must consist of one or more a's in the region before the b's. Clearly if we pump in any extra a's, we will no longer have a string in L. Thus we know that L is not regular.

Notice that we could have skipped the b's altogether and chosen w = a^Na^N. Again, we'd know that y must be a string of one or more a's. Unfortunately, if y is of even length (and it could be: remember we don't get to pick y), then we can pump in all the copies of y we want and still have a string in L. Sure, the boundary between the first half and the second half will move, that that doesn't matter. It is usually good to choose a string with a long, uniform first region followed by a definitive boundary between it and succeeding regions so that when you pump, it's clearly the first region that has changed.

(b) L = {ww : w ∈ {a, b}*}. We'll use the pumping lemma. Again, don't get confused by the use of the variable w both to define L and as the name for the string we will choose to pump on. As is always the case, the only real work we have to do is to choose an appropriate string w. We need one that is long enough (i.e., |w| ≥ N). And we need one with firm boundaries between regions. So let's choose w = a^Nba^Nb. Since |xy| ≤ N, we know that y must occur in the first a region. Clearly if we pump in any additional a's, the two halves of w will no longer be equal. Q. E. D. By the way, we could have chosen other strings for w. For example, let w = ba^Nba^N. But then there are additional choices for what y could be (since y could include the initial b) and we would have to work through them all.

(c) L = {ww' : w ∈ {a, b}*}, where w' stands for w with each occurrence of a replaced by b, and vice versa. We can prove this easily using the pumping lemma. Let w = a^Nb^N. Since |xy| ≤ N, y must be a string of all a's. So, when we pump (either in or out), we modify the first part of w but not the second part. Thus the resulting string is not in L.

We could also solve this problem just by observing that, if L is regular, so is L′ = L ∩ a*b*. But L′ is just a^nb^n, which we have already shown is not regular. Thus L is not regular either.

2. (a) L = {w : w is the unary notation for a natural number that is a multiple of 7}. L is regular since it can be described by the regular expression (1111111)*.

(b) L = {w : w is the decimal notation for a natural number that is a multiple of 7}. L is regular. We can build a deterministic FSM M to accept it. M is based on the standard algorithm for long division. The states represent the remainders we have seen so far (so there are 7 of them, corresponding to 0 – 6). The start state, of course, is 0, corresponding to a remainder of 0. So is the final state. The transitions of M are as follows:

$\forall s_i \in \{0 - 6\}$ and $\forall c_j \in \{0 - 9\}$, $\delta(s_i, c_j) = (10s_i + c_j) \bmod 7$

So, for example, on the input 962, M would first read 9. When you divide 7 into 9 you get 1 (which we don't care about since we don't actually care about the answer – we just care whether the remainder is 0) with a remainder of 2. So M will enter state 2. Next it reads 6. Since it is in state 2, it must divide 7 into 2*10 +6 (26). It gets a remainder of 5, so it goes to state 5. Next it reads 2. Since it is in state 5, it must divide 7 into 5*10 + 5 (52), producing a remainder of 3. Since 3 is not zero, we know that 862 is not divisible by 7, so M rejects.

(c) L = {w : w is the unary notation for a natural number such that there exists a pair p and q of twin primes, both > n.}. L is regular. Unfortunately, this time we don't know how to build a PDA for it. We can, however, prove that it is regular by considering the following two possibilities:

(1) There is an infinite number of twin primes. In this case, for every n, there exists a pair of twin primes greater than n. Thus L = 1*, which is clearly regular.

(2) There is not an infinite number of twin primes. In this case, there is some largest pair. There is thus also a largest n that has a pair greater than it. Thus the set of such n's is finite and so is L (the unary encodings of those values of n). Since L is finite, it is clearly regular.

It is not known which of these cases is true. But interestingly, from our point of view, it doesn't matter. L is regular in either case. It may bother you that we can assert that L is regular when we cannot draw either an FSM or a regular expression for it. It shouldn't bother you. We have just given a nonconstructive proof that L is regular (and thus, by the way, that some FSM M accepts it). Not all proofs need to be constructive. This situation isn't really any different from the case of L′ = {w : w is the unary encoding of the number of siblings I have}. You know that L′ is finite and thus regular, even though you do not know how many siblings I have and thus cannot actually build a machine to accept L′.

(d) L = {w : w is, for some $n \geq 1$, the unary notation for 10^n}. So L = {1111111111, 1^{100}, 1^{1000}, …}. L isn't regular, since clearly any machine to accept L will have to count the 1's. We can prove this using the pumping lemma: Let $w = 1^P$, $N \leq P$ and P is some power of 10. y must be some number of 1's. Clearly, it can be of length at most P. When we pump it in once, we get a string s whose maximum length is therefore 2P. But the next power of 10 is 10P. Thus s cannot be in L.

(e) L = {w : w is, for some $n \geq 1$, the decimal notation for 10^n}. Often it's easier to work with unary representations, but not in this case. This L is regular, since it is just 100*.

(f) L = {w is of the form x#y, where x, y $\in$ $\{1\}^+$ and y = x+1 when x and y are interpreted as unary numbers} (For example, 11#111 and 1111#11111 $\in$ L, while 11#11, 1#111, and 1111 $\notin$ L.) L isn't regular. Intuitively, it isn't regular because any machine to accept it must count the 1's before the # and then compare that number to the number of 1's after the #. We can prove that this is true using the pumping lemma: Let $w = 1^N\#1^{N+1}$. Since $|xy| \leq N$, y must occur in the region before the #. Thus when we pump (either in or out) we will change x but not make the corresponding change to y, so y will no longer equal x +1. The resulting string is thus not in L.

(g) L = $\{a^nb^j: |n - j| = 2\}$. L isn't regular. L consists of all strings of the form a*b* where either the number of a's is two more than the number of b's or the number of b's is two more than the number of a's. We can show that L is not regular by pumping. Let $w = a^Nb^{N+2}$. Since $|xy| \leq N$, y must equal a^p for some p > 0. We can pump y out once, which will generate the string $a^{N-p}b^{N+2}$, which is not in L.

(h) $L = \{uww^Rv : u, v, w \in \{a, b\}+\}$. L is regular. This may seem counterintuitive. But any string of length at least four with two consecutive symbols, not including the first and the last ones, is in L. We simply make everything up to the first of the two consecutive symbols u. The first of the two consecutive symbols is w. The second is w^R. And the rest of the string is v. And only strings with at least one pair of consecutive symbols (not including the first and last) are in L because w must end with some symbol s. w^R must start with that same symbol s. Thus the string will contain two consecutive occurrences of s. L is regular because it can be described the regular expression $(a \cup b)^+ (aa \cup bb) (a \cup b)^+$.

(i) $L = \{w \in \{a, b\}^* : \text{for each prefix } x \text{ of } w, \#a(x) \geq \#b(x)\}$. First we need to understand exactly what L is. In order to do that, we need to define prefix. A string x is a prefix of a string y iff $\exists z \in \Sigma^*$ such that $y = xz$. In other words, x is a prefix of y iff x is an initial substring of y. For example, the prefixes of abba are ε, a, ab, abb, and abba. So L is all strings over {a, b}* such that, at any point in the string (reading left to right), there have never been more b's than a's. The strings ε, a, ab, aaabbb, and ababa are in L. The strings b, ba, abba, and ababb are not in L. L is not regular, which we can show by pumping. Let $w = a^Nb^N$. So $y = a^p$, for some nonzero p. If we pump out, there will be fewer a's than b's in the resulting string s. So s is not in L since every string is a prefix of itself.

3. (a) Every subset of a regular language is regular. FALSE. Often the easiest way to show that a universally quantified statement such as this is false by showing a counterexample. So consider $L = a^*$. L is clearly regular, since we have just shown a regular expression for it. Now consider $L' = a^i$: i is prime. $L' \subseteq L$. But we showed in class that L′ is not regular.

(b) Let $L' = L1 \cap L2$. If L′ is regular and L2 is regular, L1 must be regular. FALSE. We know that the regular languages are closed under intersection. But it is important to keep in mind that this closure lemma (as well as all the others we will prove) only says exactly what it says and no more. In particular, it says that:

If L1 is regular and L2 is regular
Then L′ is regular.

Just like any implication, we can't run this one backward and conclude anything from the fact that L′ is regular. Of course, we can't use the closure lemma to say that L1 must not be regular either. So we can't apply the closure lemma here at all. A rule of thumb: it is almost never true that you can prove the converse of a closure lemma. So it makes sense to look first for a counterexample. We don't have to look far. Let $L' = \varnothing$. Let $L2 = \varnothing$. So L′ and L2 are regular. Now let $L1 = \{a^i: i \text{ is prime}\}$. L1 is not regular. Yet $L' = L1 \cap L2$. Notice that we could have made L2 anything at all and its intersection with ∅ would have been ∅. When you are looking for counterexamples, it usually works to look for very simple ones such as ∅ or Σ*, so it's a good idea to start there first. ∅ works well in this case because we're doing intersection. Σ* is often useful when we're doing union.

(c) If L is regular, then so is $L' = \{xy : x \in L \text{ and } y \notin L\}$. TRUE. Proof: Saying that $y \notin L$ is equivalent to saying that $y \in \overline{L}$. Since the regular languages are closed under complement, we know that $\overline{L}$ is also regular. L′ is thus the concatenation of two regular languages. The regular languages are closed under concatenation. Thus L′ must be regular.

(d) $L = \{w : w = w^R\}$ is regular. FALSE. L is NOT regular. You can prove this easily by using the pumping lemma and letting $w = a^Nba^N$.

(e) If L is a regular language, then so is $L' = \{w : w \in L \text{ and } w^R \in L\}$. TRUE. Proof: Saying that $w^R \in L$ is equivalent to saying that $w \in L^R$. If w must be in both L and L^R, that is equivalent to saying that $L' = L \cap L^R$. L is regular because the problem statement says so. L^R is also regular because the regular languages are closed under reversal. The regular languages are closed under intersection. So the intersection of L and L^R must be regular.

Proof that the regular languages are closed under reversal (by construction): If L is regular, then there exists some FSM M that accepts it. From M, we can construct a new FSM M′ that accepts L^R. M′ will effectively run M backwards. Start with the states of M′ equal to states of M. Take the state that corresponds to the start state of M and make it the final state of M′. Next we want to take the final states of M and make them the start states of M′. But M′ can have only a single start state. So create a new start state in M′ and create an epsilon transition from it to each of the states in M′ that correspond to final states of M. Now just flip the arrows on all the transitions of M and add these new transitions to M′.

(f) If C is any set of regular languages, $\cup$C is a regular language. FALSE. If C is a *finite* set of regular languages, this is true. It follows from the fact that the regular languages are closed under union. But suppose that C is an infinite set of languages. Then this statement cannot be true. If it were, then every language would be regular and we have proved that there are languages that are not regular. Why is this? Because every language is the union of some set of regular languages. Let L be an arbitrary language whose elements are w_1, w_2, w_3, …. Let C be the set of singleton languages $\{\{w_1\}, \{w_2\}, \{w_3\}, \dots\}$ such that $w_i \in L$. The number of elements of C is equal to the cardinality of L. Each individual element of C is a language that contains a single string, and so it is finite and thus regular. L = $\cup$C. Thus, since not all languages are regular, it must not be the case that $\cup$C is guaranteed to be regular. If you're not sure you follow this argument, you should try to come up with a specific counterexample. Choose an L such that L is not regular, and show that it can be described as $\cup$C for some set of languages C.

(g) $L = \{xyx^R : x, y \in \Sigma^*\}$ is regular. TRUE. Why? We've already said that xx^R isn't regular. This looks a lot like that, but it differs in a key way. L is the set of strings that can be described as some string x, followed by some string y (where x and y can be chosen completely independently), followed by the reverse of x. So, for example, it is clear that abcccccba $\in$ L (assuming $\Sigma = \{a, b, c\}$). We let x = ab, y = ccccc, and x^R = ba. Now consider abbcccccaaa. You might think that this string is not in L. But it is. We let x = a, y = bbcccccaa, and x^R = a. What about acccb? This string too is in L. We let x = ε, y = acccb, and x^R = ε. Note the following things about our definition of L: (1) There is no restriction on the length of x. Thus we can let x = ε. (2)There is no restriction on the relationship of y to x. And (3) $\varepsilon^R = \varepsilon$. Thus L is in fact equal to Σ^* because we can take any string w in Σ^* and rewrite it as ε w ε, which is of the form xyx^R. Since Σ^* is regular, L must be regular.

(h) If $L' = L1 \cup L2$ is a regular language and L1 is a regular language, then L2 is a regular language. FALSE. This is another attempt to use a closure theorem backwards. Let $L1 = \Sigma^*$. L1 is clearly regular. Since L1 contains all strings over Σ, the union of L1 with any language is just L1 (i.e., $L' = \Sigma^*$). If the proposition were true, then all languages L2 would necessarily be regular. But we have already shown that there are languages that are not regular. Thus the proposition must be false.

(i) Every regular language has a regular proper subset. FALSE. $\varnothing$ is regular. And it is subset of every set. Thus it is a subset of every regular language. However, it is not a *proper* subset of itself. Thus this statement is false. However the following two similar statements are true:

(1) Every regular language has a regular subset.
(2) Every regular language except $\varnothing$ has a regular proper subset.

(j) If L1 and L2 are nonregular languages, then $L1 \cup L2$ is also not regular. False. Let $L1 = \{a^n b^m, n \geq m\}$ and $L2 = \{a^n b^m, n \leq m\}$. $L1 \cup L2 = a^*b^*$, which is regular.

4. If L were regular, then its complement, L_1, would also be regular. L_1 contains all strings over {a, b} that are not in L. There are two ways not to be in L: have any a's that occur after any b's (in other words, not have all the a's followed by all the b's), or have an equal number of a's and b's. So now consider

$L_2 = L_1 \cap a^*b^*$

L_2 contains only those elements of L_1 in which the a's and b's are in the right order. In other words,

$L_2 = a^nb^n$

But if L were regular, then L_1 would be regular. Then L_2, since it is the intersection of two regular languages would also be regular. But we have already shown that it (a^nb^n) is not regular. Thus L cannot be regular.

5. This statement is false. To prove it, we offer a counter example. Let $L_1 = \{a^nb^m : n=m\}$ and let $L_2 = \{a^nb^m : n \neq m\}$. We have shown that both L_1 and L_2 are not regular. However,

$L_1 \cup L_2 = a^*b^*$, which is regular.

There are plenty of other examples as well. Let $L_1 = \{a^n: n \geq 1 \text{ and n is prime}\}$. Let $L_2 = \{a^n: n \geq 1 \text{ and n is not prime}\}$. Neither L_1 nor L_2 is regular. But $L_1 \cup L_2 = a^+$, which is clearly regular.

6. This is easy to prove using the pumping lemma. Let $w = a^Nba^Nba^N$. We know that xy must be contained within the first block of a's. So, no matter how y is chosen (as long as it is not empty, as required by the lemma), for any $i > 2$, $xy^iz \notin L$, since the first block of a's will be longer than the last block, which is not allowed. Therefore L is not regular.

7. First, let $L' = L \cap a^*b^*$, which must be regular if L is. We observe that $L' = a^nb^{n+2} : n \geq 0$. Now use the pumping lemma to show that L' is not regular in the same way we used it to show that a^nb^n is not regular.

8. We use the pumping lemma. Let $w = a^{2N}b^N$. xy must be contained within the block of a's, so when we pump either in or out, it will no longer be true that there will be twice as many a's as b's, since the number of a's changes but not the number of b's. Thus the pumped string will not be in L. Therefore L is not regular.

9. **(a)** L is not regular. We can prove this using the pumping lemma. Let $w = a^Nb^N$. Since y must occur within the first N characters of w, $y = a^p$ for some $p > 0$. Thus when we pump y in, we will have more a's than b's, which produces strings that are not in L.

(b) L* is also not regular. To prove this, we need first to prove a lemma, which we'll call EQAB: $\forall s, s \in L^* \Rightarrow \#a(s) = \#b(s)$. To prove the lemma, we first observe that any string s in L* must be able to be decomposed into at least one finite sequence of strings, each element of which is in L. Some strings will have multiple such decompositions. In other words, there may be more than one way to form s by concatenating together strings in L. For any string s in L*, let SQ be some sequence of elements of L that, when concatenated together, form s. It doesn't matter which one. Define the function HowMany on the elements of L*. HowMany(x) returns the length of SQ. Think of HowMany as telling you how many times we went through the Kleene star loop in deriving x. We will prove EQAB by induction on HowMany(s).

Base case: If HowMany(s) = 0, then $s = \varepsilon$. $\#a(s) = \#b(s)$.

Induction hypothesis: If HowMany(s) $\leq$ N, then $\#a(s) = \#b(s)$.

Show: If HowMany(s) = N+1, then $\#a(s) = \#b(s)$.

If HowMany(s) = N+1, then $\exists w,y$ such that $s = wy$, $w \in L^*$, HowMany(w) = N, and $y \in L$. In other words, we can decompose s into a part that was formed by concatenating together N instances of L plus a second part that is just one more instance of L. Thus we have:

(1) #a(y) = #b(y).	Definition of L	
(2) #a(w) = #b(w).	Induction hypothesis	
(3) #a(s) = #a(w) + #a(y)	s = wy	
(4) #b(s) = #b(w) + #b(y).	s = wy	
(5) #b(s) = #a(w) + #b(y)	4, 2	
(6) #b(s) = #a(w) + #a(y)	5, 1	
(7) #b(s) = #a(s)	6, 3	Q. E. D.

Now we can show that L* isn't regular using the pumping lemma. Let w = a^Nb^N. Since y must occur within the first N characters of w, y = a^p for some p > 0. Thus when we pump y in, we will have a string with more a's than b's. By EQAB, that string cannot be in L*.

CS 341 Homework 10
State Minimization

1. (a) Give the equivalence classes under $\approx_L$ for these languages:

(i) $L = (aab \cup ab)^*$

(ii) $L = \{x : x$ contains an occurrence of aababa$\}$

(iii) $L = \{xx^R : x \in \{a, b\}^*\}$

(iv) $L = \{xx : x \in \{a, b\}^*\}$

(v) $L_n = \{a, b\}a\{a, b\}^n$, where $n > 0$ is a fixed integer

(vi) The language of balanced parentheses

(b) For those languages in (a) for which the answer is finite, give a deterministic finite automaton with the smallest number of states that accepts the corresponding language.

2. Let $L = \{x \in \{a, b\}^* : x$ contains at least one a and ends in at least two b's$\}$.

(a) Write a regular expression for L.

(b) Construct a deterministic FSM that accepts L.

(c) Let R_L be the equivalence relation of the Myhill-Nerode Theorem. What partition does R_L induce on the set

$\{a, bb, bab, abb, bba, aab, abba, bbaa, baaba\}$?

(d) How many equivalence classes are there in the partition induced on Σ^* by R_L?

3. Let $L = \{x \in \{a, b\}^* : x$ begins with a and ends with b$\}$.

(a) What is the nature of the partition induced on Σ^* by R_L, the equivalence relation of the Myhill-Nerode Theorem? That is, how many classes are there in the partition and give a description of the strings in each.

(b) Using these equivalence classes, construct the minimum state deterministic FSM that accepts L.

4. Suppose that we are given a language L and a deterministic FSM M that accepts L. Assume L is a subset of $\{a, b, c\}^*$. Let R_L and R_M be the equivalence relations defined in the proof of the Myhill-Nerode Theorem. True or False:

(a) If we know that x and y are two strings in the same equivalence class of R_L, we can be sure that they are in the same equivalence class of R_M.

(b) If we know that x and y are two strings in the same equivalence class of R_M, we can be sure that they are in the same equivalence class of R_L.

(c) There must be at least one equivalence class of R_L that has contains an infinite number of strings.

(d) R_M induces a partition on $\{a, b, c\}^*$ that has a finite number of classes.

(e) If $\varepsilon \in L$, then $[\varepsilon]$ (the equivalence class containing ε) of R_L cannot be an infinite set.

5. Use the Myhill-Nerode Theorem to prove that $\{a^n b^m c^{\max(m,n)} b^p d^p : m, n, p \geq 0\}$ is not regular.

6. (a) In class we argued that the intersection of two regular languages was regular on the basis of closure properties of regular languages. We did not show a construction for the FSM that recognizes the intersection of two regular languages. Such a construction does exist, however, and it is suggested by the fact that $L_1 \cap L_2 = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$.

Given two deterministic FSMs, M_1 and M_2, that recognize two regular languages L_1 and L_2, we can construct an FSM that recognizes $L = L_1 \cap L_2$ (in other words strings that have all the required properties of both L_1 and L_2), as follows:

1. Construct machines M_1' and M_2', as deterministic versions of M_1 and M_2. This step is necessary because complementation only works on deterministic machines.
2. Construct machines M_1'' and M_2'', from M_1' and M_2', using the construction for complementation, that recognize $\Sigma^* - L_1$ and $\Sigma^* - L_2$, respectively.
3. Construct M_3, using the construction for union and the machines M_1'' and M_2'', that recognizes $((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$. This will be a nondeterministic FSM.
4. Construct M_4, the deterministic equivalent of M_3.
5. Construct M_L, using the construction for complementation, that recognizes $\Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$.

Now consider: $\Sigma = \{a, b\}$

$L_1 = \{w \in \Sigma^* :$ all a's occur in pairs$\}$ e.g., aa, aaaa, aabaa, aabbaabbb $\in L_1$

aaa, baaab, ab $\notin L_1$

$L_2 = \{w \in \Sigma^* : w \text{ contains the string bbb}\}$

Use the procedure outlined above to construct an FSM M_L that recognizes $L = L_1 \cap L_2$.

Is M_L guaranteed to be deterministic?

(b) What are the equivalence classes under $\approx_L$ for the language $L = L_1 \cap L_2$?

(c) What are the equivalence classes under $\sim_M$ for M_L in (a) above?

(d) Show how $\sim_M$ is a refinement of $\approx_L$.

(e) Use the minimization algorithm that we have discussed to construct from M_L in (a) above a minimal state machine that accepts L.

7. If you had trouble with this last one, make up another pair of L_1 and L_2 and try again.

Solutions

1. (a)

(i) L = (aab ∪ ab)*

1. [ε, aab, ab, and all other elements of L]
2. [a or wa : w ∈ L]
3. [aa or waa : w ∈ L]
4. [everything else, i.e., strings that can never become elements of L because they contain illegal substrings such as bb or aaa]

(ii) L = {x : s contains an occurrence of aababa}

1. [(a ∪ b)*aababa(a ∪ b)*, i.e., all elements of L]
2. [ε or any string not in L and ending in b but not in aab or aabab, i.e., no progress yet on "aababa"]
3. [wa for any w ∈ [2]; alternatively, any string not in L and ending in a but not in aa, aaba, or aababa]
4. [any string not in L and ending in aa]
5. [any string not in L and ending in aab]
6. [any string not in L and ending in aaba]
7. [any string not in L and ending in aabab]

Note that this time there is no "everything else". Strings never become hopeless in this language. They simply fail if we get to the end without finding "aababa".

(iii)L = $\{xx^R : x \in \{a, b\}^*\}$

1. [a, which is the only string for which the continuations that lead to acceptance are all strings of the form wa : where w ∈ L]
2. [b, which is the only string for which the continuations that lead to acceptance are all strings of the form wb : where w ∈ L]
3. [ab, which is the only string for which the continuations that lead to acceptance are all strings of the form wba : where w ∈ L]
4. [aa, which is the only string for which the continuations that lead to acceptance are all strings of the form waa : where w ∈ L]

And so forth. Every string is in a distinct equivalence class.

(iv) L = {xx : x ∈ {a, b}*}

1. [a, which is the only string for which the continuations that lead to acceptance are all strings that would be in L except that they are missing a leading a]
2. [b, which is the only string for which the continuations that lead to acceptance are all strings that would be in L except that they are missing a leading b]
3. [ab, which is the only string for which the continuations that lead to acceptance are all strings that would be in L except that they are missing a leading ab]

4. [aa, which is the only string for which the continuations that lead to acceptance are all strings that would be in L except that they are missing a leading aa]

And so forth. Every string is in a distinct equivalence class.

(v) $L_n = \{a, b\}a\{a, b\}^n$

0. [ε]
1. [a, b]
2. [aa, ba]
3. [aaa, aab, baa, bab]

.

.

.

n+2. $[(a \cup b)a(a \cup b)^n]$

n+3. [strings that can never become elements of L_n]

There is a finite number of strings in any specific language L_n. So there is a finite number of equivalence classes of $\approx_L$. Every string in L_n must be of length n+2. So there are n+3 equivalence classes (numbered 0 to n+2, to indicate the length of the strings in the class) of strings that may become elements of L_n, plus one for strings that are already hopeless, either because they don't start with ab or aa, or because they are already too long.

(vi) L = The language of balanced parentheses

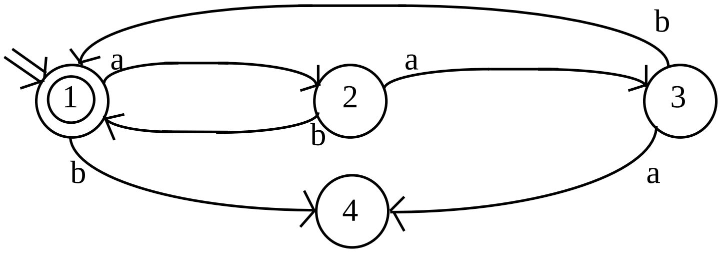
1. [w*(w*: w ∈ L] /* i.e., one extra left parenthesis somewhere in the string
2. [w*((w* : w ∈ L] /* two "
3. [w*(((w* : w ∈ L]
4. [w*((((w* : w ∈ L]
5. [w*(((((w* : w ∈ L]

… and so on. There is an infinite number of equivalence classes.

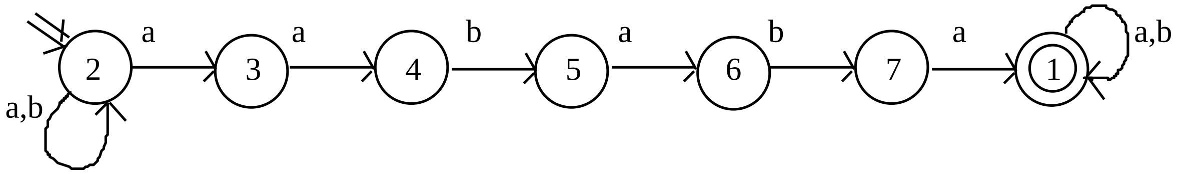
Each of these classes is distinct, since) is an acceptable continuation for 1, but none of the others;)) is acceptable for 2, but none of the others,))) is acceptable for 3, but none of the others, and so forth.

1. (b)

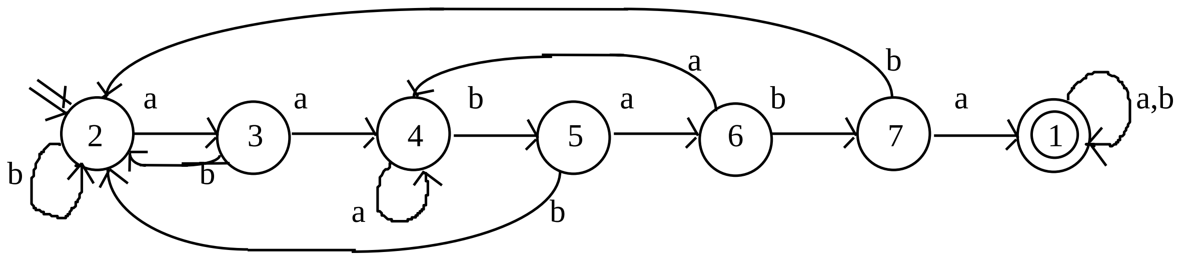
(i)



(ii) There's always a very simple nondeterministic FSM to recognize all strings that contain a specific substring. It's just a chain of states for the desired substring, with loops on all letters of Σ on the start state and the final state. In this case, the machine is:



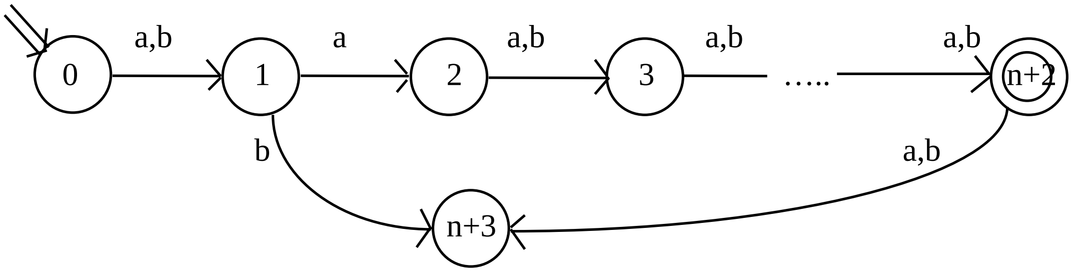
To construct a minimal, deterministic FSM, you have two choices. You can use our algorithm to convert the NDFSM to a deterministic one and then use the minimization algorithm to get the minimal machine. Or you can construct the minimal FSM directly. In any case, it is:



(iii) There is no FSM for this language, since it is not regular.

(iv) There is no FSM for this language, since it is not regular.

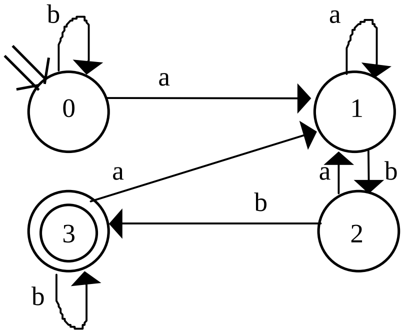
(v)



(vi) There is no FSM for this language, since it is not regular.

2. (a) $(a \cup b)^*a(a \cup b)^*bbb^*$ or $(a \cup b)^*a(a \cup b)^*\ bb$

(b)



(c) It's easiest to answer (d) first, and then to consider (c) as a special case.

(d) [0] = all strings that contain no a

[1] = all strings that end with a

[2] = all strings that end with ab

[3] = all strings that contain at least one a and that end with bb, i.e., all strings in L.

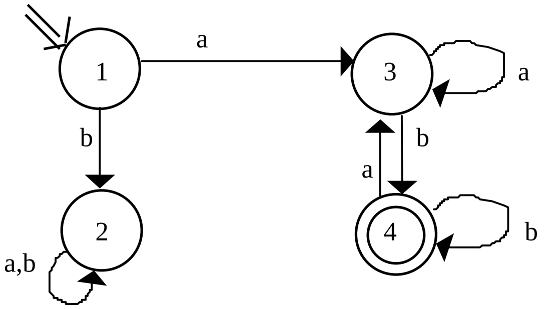
It is clear that these classes are pairwise disjoint and that their union is {a, b}*. Thus they represent a partition of {a, b}*. It is also easy to see that they are the equivalence classes of R_L of the Myhill-Nerode Theore, since all the members of one equivalence class will, when suffixed by any string z, form strings all of which are in L or all of which are not in L. Further, for any x and y from different equivalence classes, it is easy to find a z such that one of xz, yz is in L and the other is not.

Letting the equivalence relation R_L be restricted to the set in part (c), gives the partition

{{bb}, {a, bba, abba, bbaa, baaba}, {bab, aab}, {abb}}.

3. (a) $[1] = \{\varepsilon\}$
$[2] = b\ (a \cup b)^*$
$[3] = a\ \cup a(a\ \ b)^*a$
$[4] = a(a \cup b)^*b$

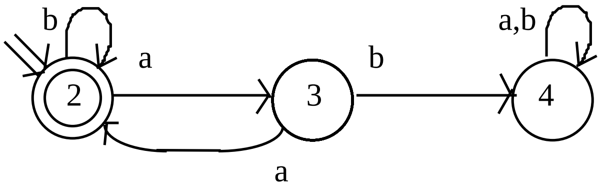
(b)



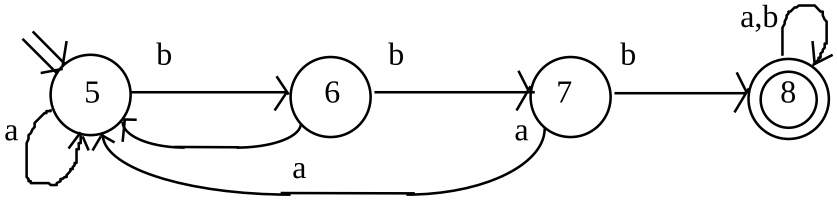
4. (a) F, **(b)** T, **(c)** T (Σ^* is infinite and the number of equivalence classes is finite), **(d)** T, **(e)** F.

5. Choose any two distinct strings of a's: call them a^i and a^j ($i < j$). Then they must be in different equivalence classes of R_L since $a^ib^ic^i \in L$ but $a^jb^ic^i \notin L$. Therefore, there is an infinite number of equivalence classes and L is not regular.

6. (a) M_1, which recognizes L_1, is:



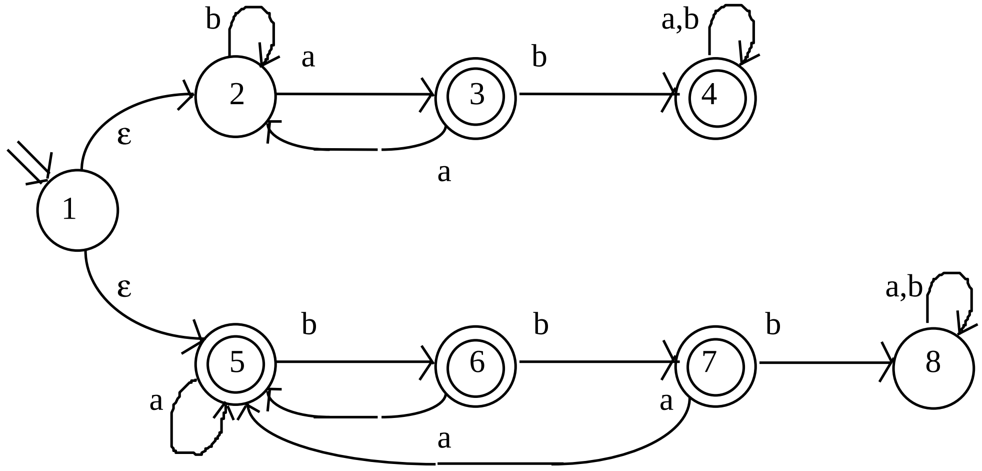
M_2, which recognizes L_2, is:



Step (1) $M_1' = M_1$ because M1 is deterministic.
$M_2' = M_2$ because M2 is deterministic.

Step (2) M_1'' is M_1' except that states 3 and 4 are the final states.
M_2'' is M_2' except that states 5, 6, and 7 are the final states.

Step (3) M_3 is:



Step (4) M_4 is:

{1, 2, 5}, a, {3, 5}
b, {2, 6}
{3, 5}, a, {2, 5}
b, {4, 6}
{2, 6}, a, {3, 5}
b, {2, 7}

{2, 5}, a, {3, 5}
b, {2, 6}
{4, 6}, a, {4, 5}
b, {4, 7}
{2, 7}, a, {3, 5}
b, {2, 8}

{4, 5}, a, {4, 5}
b, {4, 6}
{4, 7}, a, {4, 5}
b, {4, 8}
{2, 8}, a, {3, 8}
b, {2, 8}

{4, 8}, a, {4, 8}
b, {4, 8}
{3, 8}, a, {2, 8}
b, {4, 8}

s = {1, 2, 5}
F = K - {2, 8}, i.e., all states except {2, 8} are final states.

You may find it useful at this point to draw this out.

Step (5) $M_L = M_4$ except that now there is a single final state, {2, 8}.

M_L is deterministic. M_4 is deterministic by construction, and Step 5 can not introduce any nondeterminism since it doesn't introduce any transitions.

(b) 1. [strings without bbb and with any a's in pairs, including ε]
2. [strings without bbb but with a single a at the end]
3. [strings that cannot be made legal because they have a single a followed by a b]
4. [strings without bbb, with any a's in pairs, and ending in a single b]
5. [strings without bbb, with any a's in pairs, and ending with just two b's]
6. [strings with bbb and with any a's in pairs]
7. [strings with bbb but with a single a at the end]

(c) {1, 2, 5} [ε]
{3, 5} [strings without bbb but with a single a at the end]
{2, 6} [strings without bbb, with any a's in pairs, and ending in a single b]
{2, 5} [strings without bbb and with at least one pair of a's and any a's in pairs]
{4, 6} [strings that cannot be made legal because they have a single a followed by a b and where every b is preceded by an a and the last character is b]
{2, 7} [strings without bbb, with any a's in pairs, and ending with just two b's]
{4, 5} [strings that cannot be made legal because they have a single a followed by b and where there is no bbb and the last character is a]
{4, 7} [strings that cannot be made legal because they have a single a followed by a b and where there is no bbb but there is at least one bb and the last character is b]
{2, 8} [all strings in L]
{4, 8} [strings that cannot be made legal because they have a single a followed by a b and where there is a bbb, but the ab violation came before the first bbb]
{3, 8} [strings with bbb but with a single a at the end]

(d) {1, 2, 5} ∪ {2,5} = 1
{3, 5} = 2
{4, 6} ∪ {4, 5} ∪ {4, 7} ∪ {4, 8} = 3
{2, 6} = 4
{2, 7} = 5
{2, 8} = 6
{3, 8} = 7

(e) $\equiv_0$ = A: [{2, 8}],
B: [{1, 2, 5}, {2, 5}, {3, 5}, {4, 6}, {4, 5}, {4, 7}, {4, 8}, {2, 6}, {2, 7}, {3, 8}]

To compute $\equiv_1$: Consider B (since clearly A cannot be split). We need to look at all the single character transitions out of each of these states. We've already done that in Step (3) of part (a) above, so we can use that table to tell us which of our current states each state goes to. Now we just need to use that to determine which element of $\equiv_0$ they go to. We notice that all transitions are to elements of B except: ({2, 7}, b, A) and ({3, 8}, a, A). So we must split these two states from B. and they must be distinct from each other because their a and b behaviors are reversed. So we have:

$\equiv_1$ = A: [{2, 8}],
B: [{1, 2, 5}, {2, 5}, {3, 5}, {4, 6}, {4, 5}, {4, 7}, {4, 8}, {2, 6}]
C: [{2, 7}]
D: [{3, 8}]

To compute $\equiv_2$: Again we consider B:
On reading an a, all elements of B go to elements of B.
But on b: ({2, 6}, b, C), so we must split off {2, 6}. This gives us:

$\equiv_2$ = A: [{2, 8}],
B: [{1, 2, 5}, {2, 5}, {3, 5}, {4, 6}, {4, 5}, {4, 7}, {4, 8}]
C: [{2, 7}]
D: [{3, 8}]
E: [{2, 6}]

To compute $\equiv_3$: Again we consider B:
On reading an a, all elements of B go to elements of B.
But on b, {1, 2, 5} and {2, 5} go to E, while everyone else goes to B. So we have to split these two off. This gives us:

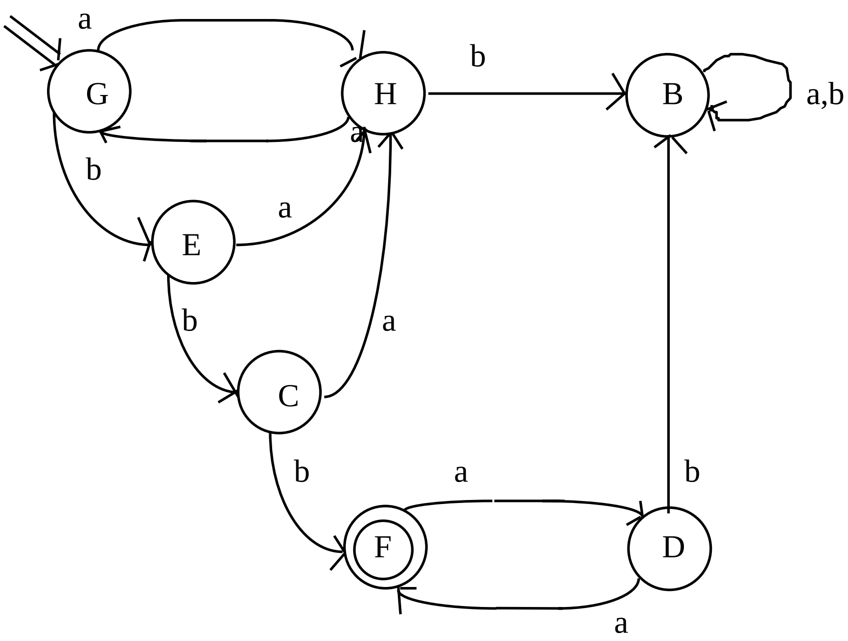
$\equiv_3$ = A: [{2, 8}],
B: [{3, 5}, {4, 6}, {4, 5}, {4, 7}, {4, 8}]
C: [{2, 7}]
D: [{3, 8}]
E: [{2, 6}]
F: [{1, 2, 5}, {2, 5}]

To compute $\equiv_4$: Again we consider B:
On reading b, all elements of B stay in B. But on reading a, {3, 5} goes to F, so we split it off. This gives us:

$\equiv_4$ = A: [{2, 8}],
B: [{4, 6}, {4, 5}, {4, 7}, {4, 8}]
C: [{2, 7}]
D: [{3, 8}]
E: [{2, 6}]
F: [{1, 2, 5}, {2, 5}]
G: [{3, 5}]

To compute $\equiv_5$: Again we consider B: On both inputs, all elements of B stay in B. So we do no further splitting, and we assert that $\equiv$ = $\equiv_4$. Notice that this is identical to what we expected from (d) above.

We can now show the minimal machine:



CS 341 Homework 11
Context-Free Grammars

1. Consider the grammar $G = (V, \Sigma, R, S)$, where

$$
\begin{aligned}
V &= \{a, b, S, A\},\\
\Sigma &= \{a, b\},\\
R = \{\quad & S \to AA,\\
& A \to AAA,\\
& A \to a,\\
& A \to bA,\\
& A \to Ab\ \ \}.
\end{aligned}
$$

(a) Which strings of $L(G)$ can be produced by derivations of four or fewer steps?

(b) Give at least four distinct derivations for the string babbab.

(c) For any $m, n, p > 0$, describe a derivation in G of the string $b^m ab^n ab^p$.

2. Construct context-free grammars that generate each of these languages:

(a) $\{wcw^R : w \in \{a, b\}^*\}$

(b) $\{ww^R : w \in \{a, b\}^*\}$

(c) $\{w \in \{a, b\}^* : w = w^R\}$

3. Consider the alphabet $\Sigma = \{a, b, (,), \cup, *, \varnothing\}$. Construct a context-free grammar that generates all strings in Σ^* that are regular expressions over $\{a, b\}$.

4. Let G be a context-free grammar and let $k > 0$. We let $L_k(G) \subseteq L(G)$ be the set of all strings that have a derivation in G with k or fewer steps.

(a) What is $L_5(G)$, where $G = (\{S, (,)\}, \{(,)\}, \{S \to \varepsilon, S \to SS, S \to (S)\ \})$?

(b) Show that, for all context-free grammars G and all $k > 0$, $L_k(G)$ is finite.

5. Let $G = (V, \Sigma, R, S)$, where

$$
\begin{aligned}
V &= \{a, b, S\},\\
\Sigma &= \{a, b\},\\
R = \{\quad & S \to aSb,\\
& S \to aSa,\\
& S \to bSa,\\
& S \to bSb,\\
& S \to \varepsilon\ \ \}.
\end{aligned}
$$

Show that $L(G)$ is regular.

6. A program in a procedural programming language, such as C or Java, consists of a list of statements, where each statement is one of several types, such as:

(1) assignment statement, of the form id := E, where E is any arithmetic expression (generated by the grammar using T and F that we presented in class).

(2) conditional statement, e.g., "if E < E then statement", or while statement , e.g. "while E < E do statement".

(3) goto statement; furthermore each statement could be preceded by a label.

(4) compound statement, i.e., many statements preceded by a begin, followed by an end, and separated by ";".

Give a context-free grammar that generates all possible statements in the simplified programming language described above.

7. Show that the following languages are context free by exhibiting context-free grammars generating each:

(a) $\{a^m b^n : m \geq n\}$

(b) $\{a^m b^n c^p d^q : m + n = p + q\}$
(c) $\{w \in \{a, b\}^* :$ w has twice as many b's as a's$\}$
(d) $\{uawb : u, w \in \{a, b\}^*, |u| = |w|\}$

8. Let $\Sigma = \{a, b, c\}$. Let L be the language of prefix arithmetic defined as follows:
(i) any member of Σ is a well-formed expression (wff).
(ii) if α and β are any wff's, then so are $A\alpha\beta$, $S\alpha\beta$, $M\alpha\beta$, and $D\alpha\beta$.
(iii) nothing else is a wff.

(One might think of A, S, M, and D as corresponding to the operators +, -, ×, /, respectively. Thus in L we could write Aab instead of the usual (a + b), and MSabDbc, instead of $((a - b) \times (b/c))$. Note that parentheses are unnecessary to resolve ambiguities in L.)

(a) Write a context-free grammar that generates exactly the wff's of L.
(b) Show that L is not regular.

9. Consider the language $L = \{a^m b^{2n} c^{3n} d^p : p > m, \text{ and } m, n \geq 1\}$.
(a) What is the shortest string in L?
(b) Write a context-free grammar to generate L.

Solutions

1. (a) We can do an exhaustive search of all derivations of length no more than 4:

$S \Rightarrow AA \Rightarrow aA \Rightarrow aa$
$S \Rightarrow AA \Rightarrow aA \Rightarrow abA \Rightarrow aba$
$S \Rightarrow AA \Rightarrow aA \Rightarrow aAb \Rightarrow aab$
$S \Rightarrow AA \Rightarrow bAA \Rightarrow baA \Rightarrow baa$
$S \Rightarrow AA \Rightarrow bAA \Rightarrow bAa \Rightarrow baa$
$S \Rightarrow AA \Rightarrow AbA \Rightarrow abA \Rightarrow aba$
$S \Rightarrow AA \Rightarrow AbA \Rightarrow Aba \Rightarrow aba$
$S \Rightarrow AA \Rightarrow Aa \Rightarrow aa$
$S \Rightarrow AA \Rightarrow Aa \Rightarrow bAa \Rightarrow baa$
$S \Rightarrow AA \Rightarrow Aa \Rightarrow Aba \Rightarrow aba$
$S \Rightarrow AA \Rightarrow AbA \Rightarrow abA \Rightarrow aba$
$S \Rightarrow AA \Rightarrow AbA \Rightarrow Aba \Rightarrow aba$
$S \Rightarrow AA \Rightarrow AAb \Rightarrow aAb \Rightarrow aab$
$S \Rightarrow AA \Rightarrow AAb \Rightarrow Aab \Rightarrow aab$

Many of these correspond to the same parse trees, just applying the rules in different orders. In any case, the strings that can be generated are: aa, aab, aba, baa.

(b) Notice that $A \Rightarrow bA \Rightarrow bAb \Rightarrow bab$, and also that $A \Rightarrow Ab \Rightarrow bAb \Rightarrow bab$. This suggests 8 distinct derivations:

$S \Rightarrow AA \Rightarrow AbA \Rightarrow AbAb \Rightarrow Abab \Rightarrow^* babbab$
$S \Rightarrow AA \Rightarrow AAb \Rightarrow AbAb \Rightarrow Abab \Rightarrow^* babbab$
$S \Rightarrow AA \Rightarrow bAA \Rightarrow bAbA \Rightarrow babA \Rightarrow^* babbab$
$S \Rightarrow AA \Rightarrow AbA \Rightarrow bAbA \Rightarrow babA \Rightarrow^* babbab$

Where each of these four has 2 ways to reach babbab in the last steps. And, of course, one could interleave the productions rather than doing all of the first A, then all of the second A, or vice versa.

(c) This is a matter of formally describing a sequence of applications of the rules in terms of m, n, p that will produce the string $b^m a b^n a b^p$.

S

$\Rightarrow$ /* by rule $S \rightarrow AA$ */

$$AA$$

$\Rightarrow^*$ /* by m applications of rule $A \rightarrow bA$ */

$$b^mAA$$

$\Rightarrow$ /* by rule $A \rightarrow a$ */

$$b^maA$$

$\Rightarrow^*$ /* by n applications of rule $A \rightarrow bA$ */

$$b^mab^nA$$

$\Rightarrow^*$ by p applications of rule $A \rightarrow Ab$ */

$$b^mab^nAb^p$$

$\Rightarrow$ /* by rule $A \rightarrow a$ */

$$b^mab^nab^p$$

Clearly this derivation (and some variations on it) produce $b^mab^nab^p$ for each m, n, p.

2. (a) $G = (V, \Sigma, R, S)$ with $V = \{S, a, b, c\}$, $\Sigma = \{a, b, c\}$, R = {

$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow c$ }.

(b) Same as (a) except remove c from V and Σ and replace the last rule, $S \rightarrow c$, by $S \rightarrow \varepsilon$.

(c) This language very similar to the language of (b). (b) was all even length palindromes; this is all palindromes. We can use the same grammar as (b) except that we must add two rules:

$S \rightarrow a$
$S \rightarrow b$

3. This is easy. Recall the inductive definition of regular expressions that was given in class :

1. ∅ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is αβ
3. If α , β are regular expressions, then so is α∪β.
4. If α is a regular expression, then so is α*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

This definition provides the basis for a grammar for regular expressions:

$G = (V, \Sigma, R, S)$ with $V = \{S, a, b, (,), \cup, *, \varnothing\}$, $\Sigma = \{ a, b, (,), \cup, *, \varnothing\}$, R= {

$S \rightarrow \varnothing$ /* part of rule 1, above
$S \rightarrow a$ /* "
$S \rightarrow b$ /* "
$S \rightarrow SS$ /* rule 2
$S \rightarrow S \cup S$ /* rule 3
$S \rightarrow S^*$ /* rule 4
$S \rightarrow (S)$ /* rule 5 }

4. (a) We omit derivations that don't produce strings in L (i.e, they still contain nonterminals).

$L_1 : S \Rightarrow \varepsilon$
$L_2 : S \Rightarrow (S) \Rightarrow ()$
$L_3 : S \Rightarrow SS \Rightarrow \varepsilon S \Rightarrow \varepsilon$
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (())$
$L_4 : S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()$
$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S) \Rightarrow ()$
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow ((()))$
$L_5 : S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()()$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())$

$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow ((((S)))) \Rightarrow ((((()))))$

So $L_5 = \{\varepsilon, (), (()), ((())), (((()))), ()() \}$

(b) We can give a (weak) upper bound on the number of strings in $L_K(G)$. Let P be the number of rules in G and let N be the largest number of nonterminals on the right hand side of any rule in G. For the first derivation step, we start with S and have P choices of derivations to take. So at most P strings can be generated. (Generally there will be many fewer, since many rules may not apply, but we're only producing an upper bound here, so that's okay.) At the second step, we may have P strings to begin with (any one of the ones produced in the first step), each of them may have up to N nonterminals that we could choose to expand, and each nonterminal could potentially be expanded in P ways. So the number of strings that can be produced is $P \times N \times P$. Note that many of them aren't strings in L since they may still contain nonterminals, but this number is an upper bound on the number of strings in L that can be produced. At the third derivation step, each of those strings may again have N nonterminals that can be expanded and P ways to expand each. In general, an upper bound on the number of strings produced after K derivation steps is $P^K N^{(K-1)}$, which is clearly finite. The key here is that there is a finite number of rules and that each rule produces a string of finite length.

5. We will show that L(G) is precisely the set of all even length strings in $\{a, b\}^*$. But we know that that language is regular. QED.

First we show that only even length strings are generated by G. This is trivial. Every rule that generates any terminal characters generates two. So only strings of even length can be generated.

Now we must show that all strings of even length are produced. This we do by induction on the length of the strings:

Base case: $\varepsilon \in L_G$ (by application of the last rule). So we generate the only string of length 0.

Induction hypothesis: All even length strings of length $\leq N$ (for even N) can be generated from S.

Induction step: We need to show that any string of length N+2 can be generated. Any string w of length $N + 2$ ($N \geq 0$) can be rewritten as xyz, where x and z are single characters and $|y| = N$. By the induction hypothesis, we know that all values of y can be generated from S. We now consider all possible combinations of values for x and z that can be used to create w. There are four, and the first four rules in the grammar generate, for any string T derivable from S, the four strings that contain T plus a single character appended at the beginning and at the end. Thus all possible strings w of length N+2 can be generated.

6. Since we already have a grammar for expressions (E), we'll just use E in this grammar and treat it as though it were a terminal symbol. Of course, what we really have to do is to combine this grammar with the one for E. As we did in our grammar for E, we'll use the terminal string id to stand for any identifier.

G = (V, Σ, R, S), where V = {S, U, C, L, T, E, :, = <, >, ;, a-z, id}, Σ = { :, = <, >, ;, a-z, id}, and

R= {

S → L U	/* a statement can be a label followed by an unlabeled statement
S → U	/* or a statement can be just an unlabeled statement. We need to make the distinction between S and U if we want to prevent a statement from being preceded by an arbitrary number of labels.
U → id := E	/* assignment statement
U → if E T E then S	/* if statement
U → while E T E do S	/* while statement
U → goto L	/* goto statement
U → begin S; S end	/* compound statement
L → id	/* a label is just an identifier

$T \rightarrow < \mid > \mid =$ /* we use T to stand for a test operator. We introduce the | (or) notation here for convenience. }

There's one problem we haven't addressed here. We have not guaranteed that every label that appears after a goto statement actually appears in the program. In general, this cannot be done with a context-free grammar.

7. (a) $L = \{a^m b^m : m \geq n\}$. This one is very similar to Example 8 in Supplementary Materials: Context-Free Languages and Pushdown Automata: Designing Context-Free Grammars. The only difference is that in that case, $m \leq n$. So you can use any of the ideas presented there to solve this problem.

(b) $L = \{a^m b^n c^p d^q : m + n = p + q\}$. This one is somewhat like **(a)**: For any string $a^m b^n c^p d^q \in L$, we will produce a's and d's in parallel for a while. But then one of two things will happen. Either $m \geq q$, in which case we begin producing a's and c's for a while, or $m \leq q$, in which case we begin producing b's and d's for a while. (You will see that it is fine that it's ambiguous what happens if m = q.) Eventually this process will stop and we will begin producing the innermost b's and c's for a while. Notice that any of those four phases could produce zero pairs. Since the four phases are distinct, we will need four nonterminals (since, for example, once we start producing c's, we do not want ever to produce any d's again). So we have:

$G = (\{S, T, U, V, a, b, c, d\}, \{a, b, c, d\}, R, S)$, where
$R = \{S \rightarrow aSd, S \rightarrow T, S \rightarrow U, T \rightarrow aTc, T \rightarrow V, U \rightarrow bUd, U \rightarrow V, V \rightarrow bVc, V \rightarrow \varepsilon\}$

Every derivation will use symbols S, T, V in sequence or S, U, V in sequence. As a quick check for fencepost errors, note that the shortest string in L is ε, which is indeed generated by the grammar. (And we do not need any rules $S \rightarrow \varepsilon$ or $T \rightarrow \varepsilon$.)

How do we know this grammar works? Notice that any string for which m = q has two distinct derivations:

$S \Rightarrow^* a^m S d^m \Rightarrow a^m T d^m \Rightarrow a^m V d^m \Rightarrow a^m b^n c^p d^q$, and
$S \Rightarrow^* a^m S d^m \Rightarrow a^m U d^m \Rightarrow a^m V d^m \Rightarrow a^m b^n c^p d^q$

Every string $a^m b^n c^p d^q \in L$ for which $m \geq q$ has a derivation:

S
$\Rightarrow$ /* by q application of rule $S \rightarrow aSd$ */
$a^q S d^q$
$\Rightarrow$ /* by rule $S \rightarrow T$ */
$a^q T d^q$
$\Rightarrow$ /* by m - q application of rule rule $T \rightarrow aTc$ */
$a^q a^{m-q} T c^{m-q} d^q = a^m T c^{m-q} d^q$
$\Rightarrow$ /* by rule $T \rightarrow V$ */
$a^m V c^{m-q} d^q$
$\Rightarrow$ /* by $n = p - (m - q)$ applications of rule $V \rightarrow bVc$ */
$a^m b^n V c^{p-(m-q)} c^{m-q} d^q = a^m b^n V c^p d^q$
$\Rightarrow$ /* by rule $V \rightarrow \varepsilon$
$a^m b^n c^p d^q$

For the other case ($m \leq q$), you can show the corresponding derivation. So every string in L is generated by G. And it can be shown that no string not in L is generated by G.

(c) $L = \{w \in \{a, b\}^* : w$ has twice as many b's as a's$\}$. This one is sort of tricky. Why? Because L doesn't care about the order in which the a's and b's occur. But grammars do. One solution is:

$G = (\{S, a, b\}, \{a, b\}, R, S)$, where $R = \{S \rightarrow SaSbSbS, S \rightarrow SbSaSbS, S \rightarrow SbSbSaS, S \rightarrow \varepsilon\}$

Try some examples to convince yourself that this grammar works. Why does it work? Notice that all the rules for S preserve the invariant that there are twice as many b's as a's. So we're guaranteed not to generate any strings that aren't in L. Now we just have to worry about generating all the strings that are in L. The first three rules handle the three possible orders in which the symbols b,b, and a can occur.

Another approach you could take is to build a pushdown automaton for L and then derive a grammar from it. This may be easier simply because PDA's are good at counting. But deriving the grammar isn't trivial either. If you had a hard time with this one, don't worry.

(d) L = {uawb : u, w ∈ {a, b}*, |u| = |w|}. This one fools some people since you might think that the a and b are correlated somehow. But consider the simpler language L' = {uaw : u, w ∈ {a, b}*, |u| = |w|}. This one seems easier. We just need to generate u and w in parallel, keeping something in the middle that will turn into a. Now back to L: L is just L' with b tacked on the end. So a grammar for L is:

G = ({S, T, a, b}, {a, b}, R, S), where R = {S → Tb, T → aTa, T → aTb, T → bTa, T → bTb, T → a}.

8. (a) G = ({S, A, M, D, F, a, b, c}. {A, M, D, S, a, b, c}, R, S), where R = {

F → a	F → AFF
F → b	F → SFF
F → c	F → MFF
	F → DFF }

(b) First, we let L' = L ∩ A*a*. L' = $\{A^n a^{n+1} : n \geq 0\}$. L' can easily be shown to be nonregular using the Pumping Theorem, so, since the regular languages are closed under intersection, L must not be regular.

9. (a) abbcccdd

(b) G = ({S, X, Y, a, b, c, d}, {a, b, c, d}, R, S), where R is either:

(S → aXdd, X → Xd, X → aXd, X → bbYccc, Y → bbYccc, Y → ε), or

(S → aSd, S → Sd, S → aMdd, M → bbccc, M → bbMccc)

CS 341 Homework 12
Parse Trees

1. Consider the grammar G = ({+, *, (,), id, T, F, E}, {+, *, (,), id}, R, E}, where
$R = \{E \rightarrow E+T,\ E \rightarrow T,\ T \rightarrow T * F,\ T \rightarrow F,\ F \rightarrow (E),\ F \rightarrow id\}$.
Give two derivations of the string id * id + id, one of which is leftmost and one of which is not leftmost.

2. Draw parse trees for each of the following:
(a) The simple grammar of English we presented in class and the string "big Jim ate green cheese."
(b) The grammar of Problem 1 and the strings id + (id + id) * id and (id * id + id * id).

3. Present a context-free grammar that generates ∅, the empty language.

4. Consider the following grammar (the start symbol is S; the alphabets are implicit in the rules):
$S \rightarrow SS \mid AAA \mid \varepsilon$
$A \rightarrow aA \mid Aa \mid b$
(a) Describe the language generated by this grammar.
(b) Give a left-most derivation for the terminal string abbaba.
(c) Show that the grammar is ambiguous by exhibiting two distinct derivation trees for some terminal string.
(d) If this language is regular, give a regular (right linear) grammar generating it and construct the corresponding FSM. If the language is not regular, prove that it is not.

5. Consider the following language : $L = \{w^R w'' : w \in \{a, b\}^*$ and w'' indicates w with each occurrence of a replaced by b, and vice versa}. Give a context-free grammar G that generates L and a parse tree that shows that aababb ∈ L.

6. (a) Consider the CFG that you constructed in Homework 11, Problem 2 for $\{wcw^R : w \in \{a, b\}^*\}$. How many derivations are there, using that grammar, for the string aabacabaa?
(b) Show parse tree(s) corresponding to your derivation(s). Is the grammar ambiguous?

7. Consider the language L = {w ∈ {a, b}* : w contains equal numbers of a's and b's}
(a) Write a context-free grammar G for L.
(b) Show two derivations (if possible) for the string aabbab using G. Show at least one leftmost derivation.
(c) Do all your derivations result in the same parse tree? If so, see if you can find other parse trees or convince yourself there are none.
(d) If G is ambiguous (i.e., you found multiple parse trees), remove the ambiguity. (Hint: look out for two recursive occurrences of the same nonterminal in the right side of a rule, e.g, $X \rightarrow XX$)
(e) See how many parse trees you get for aabbab using the grammar developed in (d).

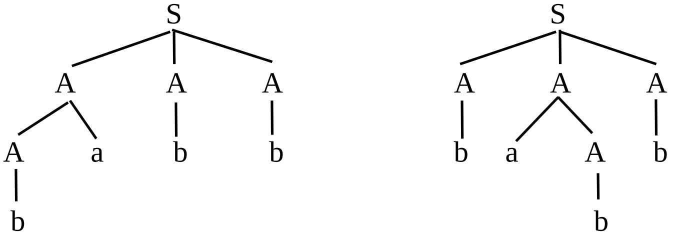
Solutions

3. G = ({S} ∪ Σ, Σ, R, S), where R is any set of rules that can't produce any strings in Σ*. So, for example, R = $\{S \rightarrow S\}$ does the trick. So does R = ∅.

4. (a) (a*ba*ba*ba*)*

(b) $S \Rightarrow AAA \Rightarrow aAAA \Rightarrow abAA \Rightarrow abAaA \Rightarrow abbaA \Rightarrow abbaAa \Rightarrow abbaba$

(c)



(d) $G = (\{S, S_1, B_1, B_2, B_3, a, b\}, \{a, b\}, R, S)$, where R = {

$S \rightarrow \varepsilon$

$S \rightarrow S_1$

$S_1 \rightarrow aS_1$

$S_1 \rightarrow bB_1$

$B_1 \rightarrow aB_1$

$B_1 \rightarrow bB_2$

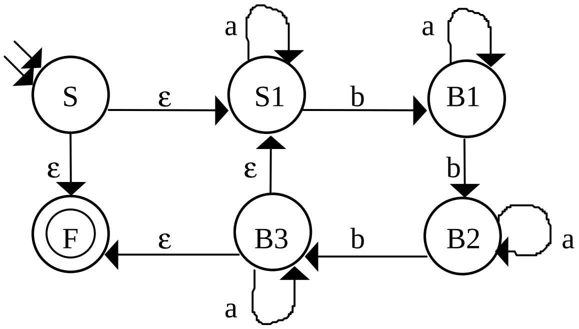
$B_2 \rightarrow aB_2$

$B_2 \rightarrow bB_3$

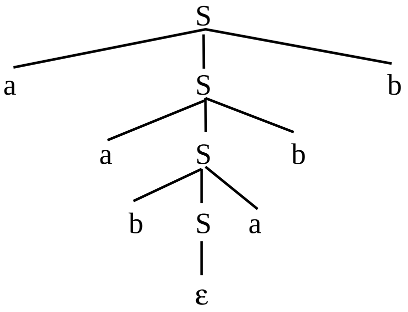
$B_3 \rightarrow aB_3$

$B_3 \rightarrow \varepsilon$

$B_3 \rightarrow S_1$



5. $G = (\{S, a, b\}, \{a, b\}, R, S)$, $R = \{ S \rightarrow aSb, S \rightarrow bSa, S \rightarrow \varepsilon\}$



6. (a) The grammar is $G = (V, \Sigma, R, S)$ with $V = \{S, a, b, c\}$, $\Sigma = \{a, b, c\}$, $R = \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow c\}$. There is a single derivation:

$S \Rightarrow aSA \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabaSabaa \Rightarrow aabacabaa$

(b) There is a single parse tree. The grammar is unambiguous.

7. (a) $G = (V, \Sigma, R, S)$ with $V = \{S \}$, $\Sigma = \{a, b \}$, R = {

$S \rightarrow aSb$

$S \rightarrow bSa$

$S \rightarrow \varepsilon$

$S \rightarrow SS$ }

(b) (i) $S \Rightarrow SS \Rightarrow aSbS \Rightarrow aaSbbS \Rightarrow aabbaSb \Rightarrow aabbab$ /* This is the leftmost derivation of the most "sensible" parse.

(ii) $S \Rightarrow SS \Rightarrow SSS \Rightarrow aSbSS \Rightarrow aaSbbSS \Rightarrow aabbSS \Rightarrow aabbaSbS \Rightarrow aabbabS \Rightarrow aabbab$ /* This is the leftmost derivation of a parse that introduced an unnecessary S in the first step, which was then eliminated by rewriting it as ε in the final step.

(c) No. The two derivations shown here have different parse trees. They do, however, have the same bracketing, [a[ab]b][ab].(In other words, they have similar essential structures.) They differ only in how S is introduced and then eliminated. But there are other derivations that correspond to additional parse trees, and some of them correspond to a completely different bracketing, [a[ab][ba]b]. One derivation that does this is

(ii) $S \Rightarrow aSb \Rightarrow aSSb \Rightarrow aabSb \Rightarrow aabbab$

(d) This is tricky. Recall that we were able to eliminate ambiguity in the case of the balanced parentheses language just by getting rid of ε except at the very top to allow for the empty string. If we do the same thing here, we get R = { $S \rightarrow \varepsilon$

$S \rightarrow T$
$T \rightarrow ab$
$T \rightarrow aTb$
$T \rightarrow ba$
$T \rightarrow bTa$
$T \rightarrow TT$

But aabbab still has multiple parses in this grammar. This language is different from balanced parens since we can go back and forth between being ahead on a's and being ahead on b's (whereas, in the paren language, we must always either be even or be ahead on open paren). So the two parses correspond to the bracketings [aabb][ab] and [a [ab] [ba] b]. The trouble is the rule T → TT, which can get applied at the very top of the tree (as in the case of the first bracketing shown here), or anywhere further down (as in the case of the second one). We clearly need some capability for forming a string by concatenating a perfectly balanced string with another one, since, without that, we'll get no parse for the string abba. Just nesting won't work. We have to be able to combine nesting and concatenation, but we have to control it. It's tempting to think that maybe an unambiguous grammar doesn't exist, but it's pretty easy to see how to build a deterministic pda (with a bottom of stack symbol) to accept this language, so there must be an unambiguous grammar. What we need is the notion of an A region, in which we are currently ahead on a's, and a B region, in which we are currently ahead on b's. Then at the top level, we can allow an A region, followed by a B region, followed by an A region and so forth. Think of switching between regions as what will happen when the stack is empty and we're completely even on the number of a's and b's that we've seen so far. For example, [ab][ba] is one A region followed by one B region. Once we enter an A region, we stay in it, always generating an a followed (possibly after something else embedded in the middle) by a b. After all, the definition of an A region, is that we're always ahead on a's. Only when we are even, can we switch to a B region. Until then, if we want to generate a b, we don't need to do a pair starting with b. We know we're ahead on a's, so make any desired b's go with an a we already have. Once we are even, we must either quit or move to a B region. If we allow for two A regions to be concatenated at the top, there will be ambiguity between concatenating two A regions at the top vs. staying in a single one. We must, however, allow two A regions to be concatenated once we're inside an A region. Consider [a[ab][ab]b] Each [ab] is a perfectly balanced A region and they are concatenated inside the A region whose boundaries are the first a and the final b. So we must distinguish between concatenation within a region (which only happens with regions of the same type, e.g, two A's within an A) and concatenation at the very top level, which only happens between different types.

Also, we must be careful of any rule of the form X → XX for another reason. Suppose we have a string that corresponds to XXX. Is that the first X being rewritten as two, or the second one being rewritten as two. We need to force a single associativity.

All of this leads to the following set of rules R:

$S \rightarrow \varepsilon$
$S \rightarrow T_a$ /* start with an A region, then optionally a B, then an A, and so forth
$S \rightarrow T_b$ /* start with a B region, then optionally an A, then a B, and so forth

$T_a \rightarrow A$ /* just a single A region
$T_a \rightarrow AB$ /* two regions, an A followed by a B
$T_a \rightarrow ABT_a$ /* we write this instead of $T_a \rightarrow T_aT_a$ to allow an arbitrary number of regions, but force associativity,
$T_b \rightarrow B$ /* these next three rules are the same as the previous three but starting with b
$T_b \rightarrow BA$
$T_b \rightarrow BAT_b$

$A \rightarrow A_1$/* this A region can be composed of a single balanced set of a's and b's
$A \rightarrow A_1A$ /* or it can have arbitrarily many such balanced sets.
$A_1 \rightarrow aAb$ /* a balanced set is a string of A regions inside a matching a, b pair
$A_1 \rightarrow ab$ /* or it bottoms out to just the pair a, b

$B \rightarrow B_1$ /* these next four rules are the same as the previous four but for B regions
$B \rightarrow B_1B$
$B_1 \rightarrow bBa$
$B_1 \rightarrow ba$

(e) The string aabbab is a single A region, and has only one parse tree in this grammar, corresponding to [[aabb][ab]]. You may also want to try abab, abba, and abaababb to see how G works.

CS 341 Homework 13
Pushdown Automata

1. Consider the pushdown automaton $M = (K, \Sigma, \Gamma, \Delta, s, F)$, where

$K = \{s, f\}$,
$F = \{f\}$,
$\Sigma = \{a, b\}$,
$\Gamma = \{a\}$,
$\Delta = \{((s, a, \varepsilon), (s, a)), ((s, b, \varepsilon), (s, a)), ((s, a, \varepsilon), (f, \varepsilon)), ((f, a, a), (f, \varepsilon)), ((f, b, a), (f, \varepsilon))\}$.

(a) Trace all possible sequences of transitions of M on input aba.
(b) Show that aba, aa, abb $\notin$ L(M), but baa, bab, baaaa $\in$ L(M).
(c) Describe L(M) in English.

2. Construct pushdown automata that accept each of the following:
(a) L = the language generated by the grammar $G = (V, \Sigma, R, S)$, where

$V = \{S, (,), [,]\}$,
$\Sigma = \{(,), [,]\}$,
$R = \{\quad S \rightarrow \varepsilon$,
$S \rightarrow SS$,
$S \rightarrow [S]$,
$S \rightarrow (S)\}$.

(b) $L = \{a^m b^n : m \le n \le 2m\}$.
(c) $L = \{w \in \{a, b\}^* : w = w^R\}$.
(d) $L = \{w \in \{a, b\}^* :$ w has equal numbers of a's and b's$\}$.
(e) $L = \{w \in \{a, b\}^* :$ w has twice as many a's as b's$\}$.
(f) $L = \{a^m b^n : m \ge n \}$
(g) $L = \{uawb: u$ and $w \in \{a, b\}^*$ and $|u| = |w|\}$

3. Consider the following language : $L = \{w^R w'' : w \in \{a, b\}^*$ and w" indicates w with each occurrence of a replaced by b, and vice versa$\}$. In Homework 12, problem 5, you wrote a context-free grammar for L. Now give a PDA M that accepts L and trace a computation that shows that aababb $\in$ L.

4. Construct a context-free grammar for the language of problem 2(b): $L = (\{a^m b^n: m \le n \le 2m\})$.

Solutions

1. (a) There are three possible computations of M on aba:

(s, aba, ε) |- (s, ba, a) |- (s, a, aa) |- (s, ε, aaa)
(s, aba, ε) |- (s, ba, a) |- (s, a, aa) |- (f, ε, aa)
(s, aba, ε) |- (f, ba, ε)

None of these is an accepting configuration.

(b) This is done by tracing the computation of M on each of the strings, as shown in (a).
(c) L(M) is the set of strings whose middle symbol is a. In other words,

$L(M) = \{xay \in \{a, b\}^* : |x| = |y|\}$.

2. (a) Notice that the square brackets and the parentheses must be properly nested. So the strategy will be to push the open brackets and parens and pop them against matching close brackets and parens as they are read in. We only need one state, since all the counting will be done on the stack. Since $\varepsilon \in L$, the start state can be final. Thus we have $M = (\{s\}, \{(,), [,]\}, \{(, [\}, \Delta, s\ \{s\})$, where (sorry about the confusing use of parentheses both as part of the notation and as symbols in the language):

Δ = {((s, (, ε), (s, ()), /* push (*/
((s, [, ε), (s, [)), /* push [*/
((s,), (),(s, ε)), /* if the input character is) and the top of the stack is (, they match */
((s,], [), (s, ε))} /* same for matching square brackets */
If we run out of input and stack at the same time, we'll accept.

(b) Clearly we need to use the stack to count the a's and then compare that count to the b's as they're read in. The complication here is that for every a, there may be either one or two b's. So we'll need nondeterminism. Every string in L has two regions, the a region followed by the b region (okay, they're hard to tell apart in the case of ε, but trivially, this even true there). So we need a machine with at least two states.

There are two ways we could deal with the fact that, each time we see an a, we don't know whether it will be matched by one b or two. The first is to push either one or two characters onto the stack. In this case, once we get to the b's, we'll pop one character for every b we see. A nondeterministic machine that follows all paths of combinations of one or two pushed characters will find at least one match for every string in L. The alternative is to push a single character for every a and then to get nondeterministic when we're processing the b's: For each stack character, we accept either one b or two. Here's a PDA that takes the second approach. You may want to try writing one that does it the other way. This machine actually needs three states since it needs two states for processing b's to allow for the case where two b's are read but only a single a is popped. So M = ({s, f, g}, {a, b}, {a}, Δ, s, {f, g}), where

Δ = { ((s, a, ε), (s, a)), /* Read an a and push one onto the stack */
((s, ε, ε), (f, ε)), /* Jump to the b reading state */
((f, b, a), (f, ε)), /* Read a single b and pop an a */
((f, b, a), (g, ε)),/* Read a single b and pop an a but get ready to read a second one */
((g, b, ε), (f, ε))}. /* Read a b without popping an a */

(c) A PDA that accepts {w : w = w^R} is just a variation of the PDA that accepts {ww^R} (which you'll find in Lecture Notes 14). You can modify that PDA by adding two transitions ((s, a, ε), (f, ε)) and ((s, b, ε), (f, ε)), which have the effect of making odd length palindromes accepted by skipping their middle symbol.

(d) We've got another counting problem here, but now order doesn't matter -- just numbers. Notice that with languages of this sort, it's almost always easier to build a PDA than a grammar, since PDAs do a good job of using their stack for counting, while grammars have to consider the order in which characters are generated. If you don't believe this, try writing a grammar for this language.

Consider any string w in L. At any point in the processing of w, one of three conditions holds: (1) We have seen equal numbers of a's and b's; (2) We have seen more a's than b's; or (3) We have seen more b's than a's. What we need to do is to use the stack to count whichever character we've seen more of. Then, when we see the corresponding instance of the other character we can "cancel" them by consuming the input and popping off the stack. So if we've seen an excess of a's, there will be a's on the stack. If we've seen an excess of b's there will be b's on the stack. If we're even, the stack will be empty. Then, whatever character comes next, we'll start counting it by putting it on the stack. In fact, we can build our PDA so that the following invariant is always maintained:

$$\text{(Number of a's read so far) - (Number of b's read so far)} \\ = \\ \text{(Number of a's on stack) - (Number of b's on stack)}$$

Notice that $w \in L$ if and only if, when we finish reading w,

$$\text{[(Number of a's read so far) - (Number of b's read so far)] = 0.}$$

So, if we build M so that it maintains this invariant, then we know that if M consumes w and ends with its stack empty, it has seen a string in L. And, if its stack isn't empty, then it hasn't seen a string in L.

To make this work, we need to be able to tell if the stack is empty, since that's the only case where we might consider pushing either a or b. Recall that we can't do that just by writing ε as the stack character, since that always matches, even if the stack is not empty. So we'll start by pushing a special character # onto the bottom of the stack. We can then check to see if the stack is empty by seeing if # is on top. We can do all the real work in our PDA in a single state. But, because we're using the bottom of stack symbol #, we need two additional states: the start state, in which we do nothing except push # and move to the working state, and the final state, which we get to once we've popped # and can then do nothing else. Considering all these issues, we get $M = (\{s, q, f\}, \{a, b\}, \{\#, a, b\}, \Delta, s, \{f\})$, where

$\Delta = \{ ((s, \varepsilon, \varepsilon), (q, \#)),$ /* push # and move to the working state q */
$((q, a, \#), (q, a\#)),$ /* the stack is empty and we've got an a, so push it */
$((q, a, a), (q, aa)),$ /* the stack is counting a's and we've got another one so push it */
$((q, b, a), (q, \varepsilon)),$ /* the stack is counting a's and we've got b, so cancel a and b */
$((q, b, \#), (q, b\#)),$ /* the stack is empty and we've got a b, so push it */
$((q, b, b), (q, bb)),$ /* the stack is counting b's and we've got another one so push it */
$((q, a, b), (q, \varepsilon)),$ /* the stack is counting b's and we've got a, so cancel b and a */
$((q, \varepsilon, \#), (f, \varepsilon))\}.$ /* the stack is empty of a's and b's. Pop the # and quit. */

To convince yourself that M does the job, you should show that M does in fact maintain the invariant we stated above.

The only nondeterminism in this machine involves the last transition in which we guess that we're at the end of the input. There is an alternative way to solve this problem in which we don't bother with the bottom of stack symbol #. Instead, we substitute a lot of nondeterminism and we sometimes push a's on top of b's, and so forth. Most of those paths will end up in dead ends. The machine has fewer states but is harder to analyze. Try to construct it if you like.

(e) This one is similar to (d) except that there are two a's for every b. Recall the two techniques for matching two to one that we discussed in our solution to (b). This time, though, we do know that there are always two a's to every b. We don't need nondeterminism to allow for either one *or* two. But, because we no longer know that all the a's come first, we do need to consider what to do in the two cases: (1) We're counting b's on the stack; and (2) We're counting a's on the stack. If we're counting b's, let's take the approach in which we push two b's every time we see one. Then, when we go to cancel a's, we can just pop one b for each a. If we see twice as many a's as b's, we'll end up with an empty stack. Now what if we're counting a's? We'll push one a for every one we see. When we see b, we pop two a's. The only special case we need to consider arises in strings such as "aba", where we'll only have seen a single a at the point at which we see the b. What we need to do is to switch from counting a's to counting b's, since the b counts twice. Thus the invariant that we want to maintain is now

(Number of a's read so far) - 2*(Number of b's read so far)
=
(Number of a's on stack) - (Number of b's on stack)

We can do all this with $M = (\{s, q, f\}, \{a, b\}, \{\#, a, b\}, \Delta, s, \{f\})$, where

$\Delta = \{ ((s, \varepsilon, \varepsilon), (q, \#)),$ /* push # and move to the working state q */
$((q, a, \#), (q, a\#)),$ /* the stack is empty and we've got an a, so push it */
$((q, a, a), (q, aa)),$ /* the stack is counting a's and we've got another one so push it */
$((q, b, aa), (q, \varepsilon)),$ /* the stack is counting a's and we've got b, so cancel aa and b */
$((q, b, a\#), (q, b\#)),$ /* the stack contains a single a and we've got b, so cancel the a and b and start counting b's, since we have a shortage of one a */
$((q, b, \#), (q, bb\#)),$ /* the stack is empty and we've got a b, so push two b's */
$((q, b, b), (q, bbb)),$ /* the stack is counting b's and we've got another one so push two */
$((q, a, b), (q, \varepsilon)),$ /* the stack is counting b's and we've got a, so cancel b and a */
$((q, \varepsilon, \#), (f, \varepsilon))\}.$ /* the stack is empty of a's and b's. Pop the # and quit. */

You should show that M preserves the invariant above.

(f) The idea here is to push each a as we see it. Then, on the first b, move to a second state and pop an a for each b. If we get to the end of the string and either the stack is empty (m = n) or there are still a's on the stack (m > n) then we accept. If we find a b and there's no a to pop, then there will be no action and so we'll fail. This machine is nondeterministic for two reasons. The first is that, in case there are no b's, we must be able to guess that we've reached the end of the input string and go pop all the a's off the stack. The second is that if there were b's but fewer than the number of a's, then we must guess that we've reached the end of the input string and pop all the a's off the stack. If we guess wrong, that path will just fail, but it will never cause us to accept something we shouldn't, since it only pops off extra a's, which is what we want anyway. We don't need a separate state for the final popping phase, since we're willing to accept either m = n or m > n. This contrasts with the example we did in class, where we required that m > n. In that case, the state where we run out of input and the stack at the same time (i.e., m = n) had to be a rejecting state. Thus we needed an additional accepting state where we popped the stack.

M = ({1, 2}, {a, b}, {a}, 1, {2}, Δ =

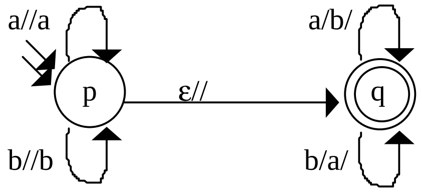
((1, a, ε), (1, a)) /* push an a on the stack for every input a
((1, b, a), (2, ε)) /* pop an a for the first b and go to the b-popping state
((1, ε, ε), (2, ε) /* in case there aren't any b's -- guess end of string and go pop any a's
((2, b, a), (2, ε) /* for each input b, pop an a off the stack
((2, ε, a), (2, ε) /* if we run out of input while there are still a's on the stack,
then pop the a's and accept

(g) The idea here is to create a nondeterministic machine. In the start state (1), it reads a's and b's, and for each character seen, it pushes an x on the stack. Thus it counts the length of u. If it sees an a, it may also guess that this is the required separator a and go to state 2. In state 2, it reads a's and b's, and for each character seen, pops an x off the stack. If there's nothing to pop, the machine will fail. If it sees a b, it may also guess that this is the required final b and go to the final state, state 3. The machine will then accept if both the input and the stack are empty.

M = ({1, 2, 3}, {a, b}, {x}, s, {2}, Δ =

((1, a, ε), (1, x)) /* push an x on the stack for every input a
((1, b, ε), (1, x)) /* push an x on the stack for every input b
((1, a, ε), (2, ε)) /* guess that this is the separator a. No stack action
((2, a, x), (2, ε) /* for each input a, pop an x off the stack
((2, b, x), (2, ε) /* for each input b, pop an x off the stack
((2, b, ε), (3, ε) /* guess that this is the final b and go to the final state

3.



4. S → ε
S → aSb
S → aSbb

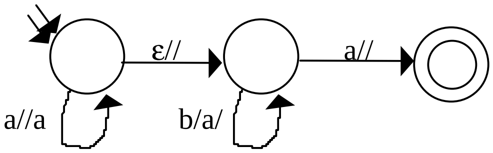
CS 341 Homework 14
Pushdown Automata and Context-Free Grammars

1. In class, we described an algorithm for constructing a PDA to accept a language L, given a context free grammar for L. Let L be the balanced brackets language defined by the grammar G = ({S, [,]}, {[,]}, R, S), where R =

$S \to \varepsilon$, $S \to SS$, $S \to [S]$

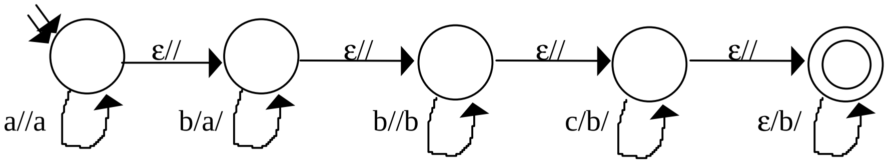
Apply the construction algorithm to this grammar to derive a PDA that accepts L. Trace the operation of the PDA you have constructed on the input string [[][]].

2. Consider the following PDA M:



(a) What is L(M)?
(b) Give a deterministic PDA that accepts L(M) (***not*** L(M)\$).

3. Write a context-free grammar for L(M), where M is



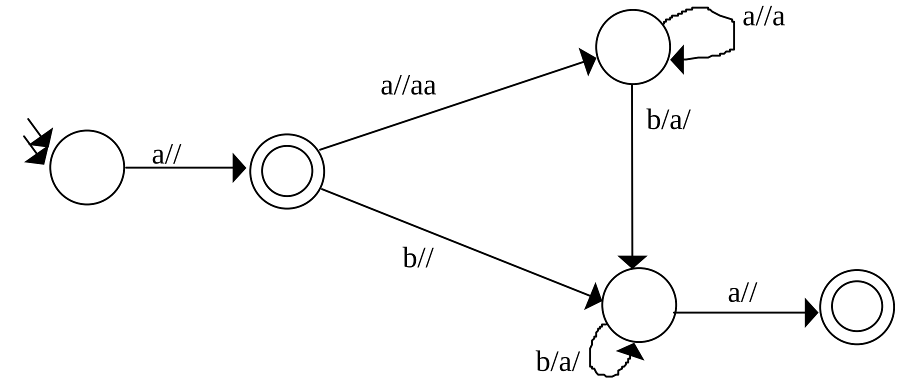
4. Consider the language L = $\{ba^{m1}ba^{m2}b\ldots ba^{mn} : n \geq 2, m1, \ldots, mn \geq 0$, and $mi \neq mj$ for some $i, j\}$
(a) Give a nondeterministic PDA that accepts L.
(b) Write a context-free grammar that generates L.
(c) Prove that L is not regular.

Solutions

1. This is a very simple mechanical process that you should have no difficulty carrying out, and getting the following PDA, M = ({p, q}, {[,]}, {S, [,]}, Δ, p, {q}), where

Δ = {((p, ε, ε), (q, S)),
((q, ε, S), (q, ε)), ((q, ε, S), (q, SS)), ((q, ε, S), (q, [S])),
((q, [, [), (q, ε)), ((q,],]), (q, ε))}

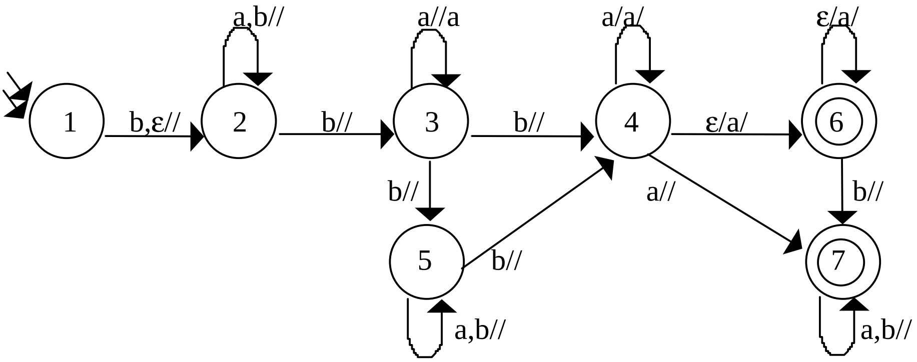
2. (a) L(M) = $\{a^n b^n a : n \geq 0\}$
(b)



3. Don't even try to use the grammar construction algorithm. Just observe that L = $\{a^n b^n b^m c^p : m \geq p$ and n and $p \geq 0\}$, or, alternatively $\{a^n b^m c^p : m \geq n + p$ and n and $p \geq 0\}$. It can be generated by the following rules:

$S \rightarrow S_1S_2$
$S_1 \rightarrow aS_1b$ /* S_1 generates the a^nb^n part. */
$S_1 \rightarrow \varepsilon$
$S_2 \rightarrow bS_2$ /* S_2 generates the b^mc^p part. */
$S_2 \rightarrow bS_2c$
$S_2 \rightarrow \varepsilon$

4. (a)



We use state 2 to skip over an arbitrary number of ba^i groups that aren't involved in the required mismatch.
We use state 3 to count the first group of a's we care about.
We use state 4 to count the second group and make sure it's not equal to the first.
We use state 5 to skip over an arbitrary number of ba^i groups in between the two we care about.
We use state 6 to clear the stack in the case that the second group had fewer a's than the first group did.
We use state 7 to skip over any remaining ba^i groups that aren't involved in the required mismatch.

(b) $S \rightarrow A'bLA'$ /* L will take care of two groups where the first group has more a's */
$S \rightarrow A'bRA'$ /* R will take care of two groups where the second group has more a's */
$L \rightarrow ab \mid aL \mid aLa$
$R \rightarrow ba \mid Ra \mid aRa$
$A' \rightarrow bAA' \mid \varepsilon$
$A \rightarrow aA \mid \varepsilon$

(c) Let $L_1 = ba^*ba^*$, which is obviously regular.
If L is regular then
$L_2 = L \cap L_1$ is regular.
$L_2 = ba^nba^m, n \neq m$
$\neg L_2 \cap L_1$ must also be regular.
But $\neg L_2 \cap L_1 = ba^nba^m, n = m$, which can easily be shown, using the pumping theorem, not to be regular.

CS 341 Homework 15
Parsing

1. Show that the following languages are deterministic context free.

(a) $\{a^mb^n : m \neq n\}$

(b) $\{wcw^R : w \in \{a, b\}^*\}$

(c) $\{ca^mb^m : m \geq 0\} \cup \{da^mb^{2m} : m \geq 0\}$

(d) $(a^mcb^m : m \geq 0\} \cup \{a^mdb^{2m} : m \geq 0\}$

2. Consider the context-free grammar: G = (V, Σ, R, S), where V = {(,), ., a, S, A}, Σ = {(,), .}, and R =

{ S → (),
S → a,
S → (A),
A → S,
A → A.S}

(If you are familiar with the programming language LISP, notice that L(G) contains all atoms and lists, where the symbol a stands for any non-null atom.)

(a) Apply left factoring and the rule for getting rid of left recursion to G. Let G' be the resulting grammar. Argue that G' is LL(1). Construct a deterministic pushdown automaton M that accepts L(G)$ by doing a top down parse. Study the computation of M on the string ((()).a).

(b) Repeat Part (a) for the grammar resulting from G if one replaces the first rule by A → ε.

(c) Repeat Part (a) for the grammar resulting from G if one replaces the last rule by A → S.A.

3. Answer each of the following questions True or False. If you choose false, you should be able to state a counterexample.

(a) If a language L can be described by a regular expression, we can be sure it is a context-free language.

(b) If a language L cannot be described by a regular expression, we can be sure it is not a context-free language.

(c) If L is generated by a context-free grammar, then L cannot be regular.

(d) If there is no pushdown automaton accepting L, then L cannot be regular.

(e) If L is accepted by a nondeterministic finite automaton, then there is some deterministic PDA accepting L.

(f) If L is accepted by a deterministic PDA, then L' (the complement of L) must be regular.

(g) If L is accepted by a deterministic PDA, then L' must be context free.

(h) If, for a given L in {a, b}*, there exist x, y, z, such that $y \neq \varepsilon$ and $xy^nz \in L$ for all $n \geq 0$, then L must be regular.

(i) If, for a given L in {a, b}*, there do not exist u, v, x, y, z such that $|vy| \geq 1$ and $uv^nxy^nz \in L$ for all $n \geq 0$, then L cannot be regular.

(j) If L is regular and L = L1 ∩ L2 for some L1 and L2, then at least one of L1 and L2 must be regular.

(k) If L is context free and L = L1L2 for some L1 and L2, then L1 and L2 must both be context free.

(l) If L is context free, then L* must be regular.

(m) If L is an infinite context-free language, then in any context-free grammar generating L there exists at least one recursive rule.

(n) If L is an infinite context-free language, then there is some context-free grammar generating L that has no rule of the form A → B, where A and B are nonterminal symbols.

(o) Every context-free grammar can be converted into an equivalent regular grammar.

(p) Given a context-free grammar generating L, every string in L has a right-most derivation.

4. Recall problem 4 from Homework 12. It asked you to consider the following grammar for a language L (the start symbol is S; the alphabets are implicit in the rules):

S → SS | AAA | ε
A → aA | Aa | b

(a) It is not possible to convert this grammar to an equivalent one in Chomsky Normal Form. Why not?

(b) Modify the grammar as little as possible so that it generates L - ε. Now convert this new grammar to Chomsky Normal Form. Is the resulting grammar still ambiguous? Why or why not?

(c) From either the original grammar for L - ε or the one in Chomsky Normal Form, construct a PDA that accepts L - ε.

5. Consider the following language : L = {$w^R w''$: w ∈ {a, b}* and w'' indicates w with each occurrence of a replaced by b, and vice versa}. In Homework 12, problem 5, you wrote a context-free grammar for L. Then, in Homework 13, problem 3, you wrote a PDA M that accepts L and traced one of its computations. Now decide whether you think L is deterministic context free. Defend your answer.

6. Convert the following grammar for arithmetic expressions to Chomsky Normal Form:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

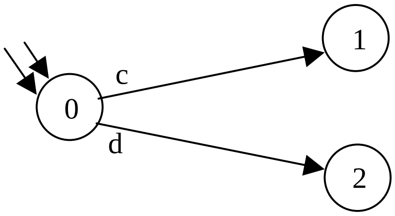
7. Again, consider the grammar for arithmetic expressions given in Problem 6. Walk through the process of doing a top down parse of the following strings using that grammar. Point out the places where a decision has to be made about what to do.
(a) id * id + id
(b) id * id * id

Solutions

1. (a) L = {$a^m b^n$: m ≠ n}. To show that a language L is deterministic context free, we need to show a deterministic PDA that accepts L$. We did that for L = {$a^m b^n$: m ≠ n} in class. (See Lecture Notes 14).

(b) L = {wcw^R : w ∈ {a, b}*}. In class (again see Lecture Notes 14), we built a deterministic PDA to accept L = {wcw^R : w ∈ {a, b}*}. It's easy to turn it into a deterministic PDA that accepts L$.

(c) L = {$ca^m b^m$: m ≥ 0} ∪ {$da^m b^{2m}$: m ≥ 0}. Often it's hard to build a deterministic PDA for a language that is formed by taking the union of two other languages. For example, {$a^m b^m$: m ≥ 0} ∪ {$a^m b^{2m}$: m ≥ 0} would be hard (in fact it's impossible) because we have no way of knowing, until we run out of b's, whether we're expecting two b's for each a or just one. However, {$ca^m b^m$: m ≥ 0} ∪ {$da^m b^{2m}$: m ≥ 0} is actually quite easy. Every string starts with a c or a d. If it's a c, then we know to look for one b for each a; if it's a d, then we know to look for two. So the first thing we do is to start our machine like this:



The machine that starts in state 1 is our classic machine for $a^n b^n$, except of course that it must have a final transition on $ to its final state.

We have two choices for the machine that starts in state 2. It can either push one a for every a it sees, and then pop an a for every pair of b's, or it can push two a's for every a it sees, and then pop one a for every b.

(d) $L = (a^m cb^m : m \geq 0\} \cup \{a^m db^{2m} : m \geq 0\}$. Now we've got another unioned language. But this time we don't get a clue from the first character which part of the language we're dealing with. That turns out to be okay though, because we do find out before we have to start processing the b's whether we've got two b's for each a or just one. Recall the two approaches we mentioned for machine 2 in the last problem. What we need here is the first, the one that pushes a single a for each a it sees. Then, when we see a c or d, we branch and either pop an a for each b or pop an a for every two b's.

2. (a) We need to apply left factoring to the two rules $S \rightarrow ()$ and $S \rightarrow (A)$. We also need to eliminate the left recursion from $A \rightarrow A\ .\ S$. Applying left factoring, we get the first column shown here. Then getting rid of left recursion gets us the second column:

$S \rightarrow (S'$	$S \rightarrow (S'$
$S' \rightarrow)$	$S' \rightarrow)$
$S' \rightarrow A)$	$S' \rightarrow A)$
$S \rightarrow a$	$S \rightarrow a$
$A \rightarrow S$	$A \rightarrow SA'$
$A \rightarrow A.S$	$A' \rightarrow .SA'$
	$A' \rightarrow \varepsilon$

(b) Notice that the point of the first rule, which was $S \rightarrow ()$, was to get a set of parentheses with no A inside. An alternative way to do that is to dump that rule but to add the rule $A \rightarrow \varepsilon$. Now we always introduce an A when we expand S, but we can get rid of it later. If we do this, then there's no left factoring to be done. We still have to get rid of the left recursion on A, just as we did above, however.

(c) If we change $A \rightarrow A.S$ to $S \rightarrow S.A$, then there's no left recursion to get rid of and we can leave the rules unchanged. Notice, though, that we'll get different parse trees this way, which may or may not be important. To see this, consider the string (a.a.a) and parse it using both the original grammar and the one we get if we change the last rule.

3. (a) True, since all regular languages are context-free.

(b) False, there exist languages that are context-free but not regular.

(c) False. All regular languages are also context-free and thus are generated by context-free grammars.

(d) True, since if L were regular, it would also be context free and thus would be accepted by some PDA.

(e) True, since there must also be a deterministic FSM and thus a deterministic PDA.

(f) False. Consider $L = a^n b^n$. $L' = \{w \in \{a, b\}^* :$ either some b comes before some a or there is an unequal number of a's and b's.$\}$. Clearly this language is not regular since we can't count the a's and b's.

(g) True, since the deterministic context-free languages are closed under complement.

(h) False. Suppose $L = a^n c^* b^n$, which is clearly not regular. Let x = aa, y = c, and z = bb. $xy^n z \in L$.

(i) False. L could be finite.

(j) False. L1 could be $a^n b^n$ and L2 could be $\{\varepsilon \cup a^n b^m : n \neq m\}$. Neither is regular. But $L1 \cap L2 = \{\varepsilon\}$, which is regular.

(k) False. Let $L1 = a^*$ and $L2 = \{a^n b^m c^m : n \neq m\}$. L2 is not context free. But $L = L1L2 = a^* b^m c^m$, which is context free.

(l) False. Let $L = ww^R$.

(m) True.

(n) True, since we have a procedure for eliminating such unit productions.

(o) False, since there exist context-free languages that are not regular.

(p) True.

4. (a) No grammar in Chomsky Normal Form can generate ε, yet $\varepsilon \in L$.

(b) In the original grammar, we could generate zero copies of AAA (by letting S go to ε), one copy of AAA (by letting S go to AAA), two copies (by letting S go to SS and then each of them to AAA), three copies of AAA (by letting S go to SS, then one of the S's goes to SS, then all three go to AAA), and so forth. We want to make sure that we can still get one copy, two copies, three copies, etc. We just want to eliminate the zero copies option. Note that the only role of S is to determine how many copies of AAA are produced. Once we have generated A's we can never go back and apply the S rules again. So all we have to do is to eliminate the production S → ε. The modified grammar to accept L - ε is thus:

G = ({S, A, B, C, a, b}, {a, b}, R, S), where R = {
S → SS | AAA
A → aA | Aa | b

If we convert this grammar to Chomsky Normal Form, we get:

G = ({S, A, B, C, a, b}, {a, b}, R, S), where R = {

S → SS	A → AC
S → AB	A → b
B → AA	C → a
A → CA }	

This grammar is still ambiguous.

(c) (from the grammar of part (b)): M = ({p, q}, {a, b}, {S, A, a, b,}, Δ, p, {q})

Δ = { ((p, ε, ε), (q, S))
((q, ε, S), (q, SS)
((q, ε, S), (q, AAA)
((q, ε, A), (q, aA))
((q, ε, A), (q, Aa))
((q, ε, A), (q, b))
((q, a, a), (q, ε))
((q, b, b), (q, ε)) }

5. L is not deterministic context free for essentially the same reason that ww^R is not.

6. The original grammar was:
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2. There are no ∊ rules. We show steps 3, 4, and 5 next to each other, so it's clear where the rules in steps 4 and 5 came from. In each case, the first rule that is derived from a step 3 rule is next to its source. If more than one rule is derived from any given step 3 rule, the second and others are shown immediately under the first. That's why there are some blank lines in the first two columns.

Step 3.

E ⇒* T, F
T ⇒* F

G" = E → E + T

T → T * F

F → (E)

F → id

Then we add:

E → T * F
E → (E)
E → id
T → (E)
T → id

Step 4.

E → EPT

T → TMF

F → LER

F → id

E → TMF
E → LER
E → id
T → LER
T → id
P → +
M → *
L → (
R →)

Step 5.

E → E E'
E' → PT
T → T T'
T' → M F
F → L F'
F' → E R
F → id

E →T T' (since T' → M F)
E → LF' (since F' → E R)
E → id
T → L F' (since F' → E R)
T → id
P → +
M → *
L → (
R →)

CS 341 Homework 16
Languages that Are and Are Not Context-Free

1. Show that the following languages are context-free. You can do this by writing a context free grammar or a PDA, or you can use the closure theorems for context-free languages. For example, you could show that L is the union of two simpler context-free languages.

(a) L = a^ncb^n

(b) L = $\{a, b\}^* - \{a^nb^n : n \geq 0\}$

(c) L = $\{a^mb^nc^pd^q : n = q, \text{ or } m \leq p \text{ or } m + n = p + q\}$

(d) L = $\{a, b\}^* - L_1$, where L_1 is the language $\{babaabaaab\ldots ba^{n-1}ba^nb : n\ \ n \geq 1\}$.

2. Show that the following languages are not context-free.

(a) L = $\{ a^{n^2} : n \geq 0\}$

(b) L = $\{www : w \in \{a, b\}^*\}$

(c) L = $\{w \in \{a, b, c\}^* :$ w has equal numbers of a's, b's, and c's$\}$

(d) L = $\{a^nb^ma^n : n \geq m\}$

(e) L = $\{a^nb^mc^nd^{(n+m)} : m, n \geq 0\}$

3. Give an example of a context free language ($\neq \Sigma^*$) that contains a subset that is not context free. Describe the subset.

4. What is wrong with the following "proof" that $a^nb^{2n}a^n$ is context free?

(1) Both $\{a^nb^n : n \geq 0\}$ and $\{b^na^n : n \geq 0\}$ are context free.

(2) $a^nb^{2n}a^n = \{a^nb^n\}\{b^na^n\}$

(3) Since the context free languages are closed under concatenation, $a^nb^{2n}a^n$ is context free.

5. Consider the following context free grammar: G = ({S, A, a, b}, {a, b}, R, S), where R = {

$S \rightarrow aAS$
$S \rightarrow a$
$A \rightarrow SbA$
$A \rightarrow SS$
$A \rightarrow ba$ }

(a) Answer each of the following questions True or False:

(i) From the fact that G is context free, it follows that there is no regular expression for L(G).

(ii) L(G) contains no strings of length 3.

(iii) For any string $w \in L(G)$, there exists u, v, x, y, z such that w = uvxyz, $|vy| \geq 1$, and $uv^nxy^nz \in L(G)$ for all $n \geq 0$.

(iv) If there exist languages L1 and L2 such that L(G) = L1 $\cup$ L2, then L1 and L2 must both be context free.

(v) The language $(L(G))^R$ is context free.

(b) Give a leftmost derivation according to G of aaaabaa.

(c) Give the parse tree corresponding to the derivation in (b).

(d) Give a nondeterministic PDA that accepts L(G).

6. Show that the following language is context free:

L = $\{xx^Ryy^Rzz^R : x, y, z \in \{a, b\}^*\}$.

7. Suppose that L is context free and R is regular.

(a) Is L - R necessarily context free?

(b) Is R - L necessarily context free?

8. Let $L_1 = \{a^nb^m : n \geq m\}$. Let $R_1 = \{(a \cup b)^* :$ there is an odd number of a's and an even number of b's$\}$. Show a pda that accepts $L_1 \cap R_1$.

Solutions

1. **(a)** $L = a^ncb^n$. We can easily do this one by building a CFG for L. Our CFG is almost the same as the one we did in class for a^nb^n:

$S \rightarrow aSB$
$S \rightarrow c$

(b) $L = \{a, b\}^* - \{a^nb^n : n \geq 0\}$. In other words, we've got the complement of a^nb^n. So we look at how a string could fail to be in a^nb^n. There are two ways: either the a's and b's are out of order or there are not equal numbers of them. So our language L is the union of two other languages:

- $L_1 = (a \cup b)^* - a^*b^*$ (strings where the a's and b's are out of order)
- $L_2 = a^nb^m\ n \neq m$ (strings where the a's and b's are in order but there aren't matching numbers of them)

L_1 is context free. We gave a context-free grammar for it in class (Lecture 12). L_2 is the complement of the regular language a^*b^*, so it is also regular and thus context free. Since the union of two context-free languages is context free, L is context free.

(c) $L = \{a^mb^nc^pd^q : n = q$ or $m \leq p$ or $m + n = p + q\}$. This one looks scary, but it's just the union of three quite simple context-free languages:

$L_1 = a^mb^nc^pd^q : n = q$
$L_2 = a^mb^nc^pd^q : m \leq p$
$L_3 =a^mb^nc^pd^q : m + n = p + q$
You can easily write context-free grammars for each of these languages.

(d) $L = \{a, b\}^* - L_1$, where L_1 is the language $\{babaabaaab\ldots ba^{n-1}ba^nb : n\ \ n \geq 1\}$. This one is interesting. L_1 is not context free. But its complement L is. There are two ways to show this:

1. We could build a PDA. We can't build a PDA for L_1: if we count the first group of a's then we'll need to pop them to match against the second. But then what do we do for the third? L is easier though. We don't need to check that all the a groups are right. We simply have to find one that is wrong. We can do that with a nondeterministic pda P. P goes through the string making sure that the basic $b(a^+b)^+$ structure is followed. Also, it nondeterministically chooses one group of a's to count. It then checks that the following group does not contain one more a. If any branch succeeds (i.e., it finds a mismatched pair of adjacent a groups) then P accepts.
2. We could use the same technique we used in (b) and (c) and decompose L into the union of two simpler languages. Just as we did in (b), we ask how a string could fail to be in L_1 and thus be in L. The answer is that it could fail to have the correct $b(a^+b)^+$ structure or it could have regions of a's that don't follow the rule that each region must contain one more a than did its predecessor. Thus L is the union of two languages:
   - $L_2 = (a \cup b)^* - b(a^+b)^+$
   - $L_3 = \{xba^mba^nby \in \{a, b\}^* : m+1 \neq n\}$.

   It's easy to show that L_2 is context free: Since $b(a^+b)^+$ is regular its complement is regular and thus context free. L_3 is also context free. You can build either a CFG or a PDA for it. It's very similar to the simpler language $a^nb^m\ n \neq m$ that we used in (b) above. So $L_1 = L_2 \cup L_3$ is context free.

2. **(a)** $L = \{ a^{n^2} : n \geq 0\}$. Suppose $L = \{ a^{n^2} : n \geq 0\}$ were context free. Then we could pump. Let $n = M^2$. So w is the string with M^{2^2} , or M^4, a's.) Clearly $|w| \geq K$, since $M > K$. So uvvxyyz must be in L (whatever v and y are). But it can't be. Why not? Given our w, the next element of L is the string with $(M^2+1)^2$ a's. That's $M^4 + 2M^2 + 1$ (expanding it out). But we know that $|vxy| \leq M$, so we can't pump in more than M a's when we pump only once. Thus the string we just created can't have more than $M^4 + M$ a's. Clearly not enough.

(b) L = **{**www : w ∈ {a, b}*}. The easiest way to do this is not to prove directly that L = **{**www : w ∈ {a, b}*} is not context free. Instead, let's consider L1 = L ∩ a*ba*ba*b. If L is context free, L1 must also be. L1 = $\{a^nba^nba^nb : n \geq 0\}$. To show that L1 is not context free, let's choose $w = a^Mba^Mba^Mb$. First we observe that neither v nor y can contain b, because if either did, then, when we pump, we'd have more than three b's, which is not allowed. So both must be in one of the three a regions. We consider the cases:

(1, 1) That group of a's will no longer match the other two, so the string is not in L1.
(2, 2) "
(3, 3) "
(1, 2) At least one of these two groups will have something pumped into it and will no longer match the one that is left out.
(2, 3) "
(1, 3) excluded since $|vxy| \leq M$, so vxy can't span the middle region of a's.

(c) L = {w ∈ {a, b, c}*. Again, the easiest thing to do is first to intersect L = {w ∈ {a, b, c}* : w has equal numbers of a's, b's, and c's} with a regular language. This time we construct L1 = L ∩ a*b*c*. L1 must be context free if L is. But L1 = $a^nb^nc^n$, which we've already proven is not context free. So L isn't either.

(d) L = $\{a^nb^ma^n : n \geq m\}$. We'll use the pumping lemma to show that L = $\{a^nb^ma^n : n \geq m\}$ is not context free. Choose $w = a^Mb^Ma^M$. We know that neither v nor y can cross a and b regions, because if one of them did, then, when we pumped, we'd get a's and b's out of order. So we need only consider the cases where each is in one of the three regions of w (the first group of a's, the b's, and the second group of a's.)

(1, 1) The first group of a's will no longer match the second group.
(2, 2) If we pump in b's, then at some point there will be more b's than a's, and that's not allowed.
(3, 3) Analogous to (1, 1)
(1, 2) We must either (or both) pump a's into region 1, which means the two a regions won't match, or, if y is not empty, we'll pump in b's but then eventually there will be more b's than a's.
(2, 3) Analogous to (1, 2)
(1, 3) Ruled out since $|vxy| \leq M$, so vxy can't span the middle region of b's.

(e) L = $\{a^nb^mc^nd^{(n+m)} : m, n \geq 0\}$. We can show that L = $\{a^nb^mc^nd^{(n+m)} : m, n \geq 0\}$ is not context free by pumping. We choose $w = a^Mb^Mc^Md^{2M}$. Clearly neither v nor y can cross regions and include more than one letter, since if that happened we'd get letters out of order when we pumped. So we only consider the cases where v and y fall within a single region. We'll consider four regions, corresponding to a, b, c, and d.

(1, 1) We'll change the number of a's and they won't match the c's any more.
(1, 2) If v is not empty, we'll change the a's and them won't match the c's. If y is nonempty, we'll change the number of b's and then we won't have the right number of d's any more.
(1, 3), (1, 4) are ruled out because $|vxy| \leq M$, so vxy can't span across any whole regions.
(2, 2) We'll change the number of b's but then we won't have the right number of d's.
(2, 3) If v is not empty, we'll change the b's without changing the d's. If y is not empty, we'll change the c's and they'll no longer match the a's.
(2, 4) is ruled out because $|vxy| \leq M$, so vxy can't span across any whole regions.
(3, 3) We'll change the number of c's and they won't match the a's.
(3, 4) If v is not empty, we'll change c's and they won't match a's. If y is not empty, we'll change d's without changing b's.
(4, 4) We'll change d's without changing a's or b's.

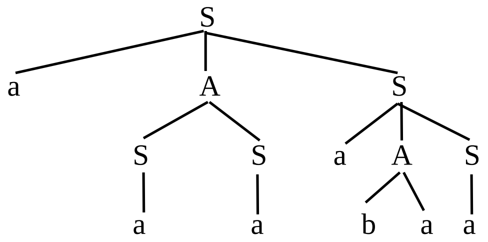
3. Let L = { $a^nb^mc^p$: n = m or m = p}. L is clearly context free. We can build a nondeterministic PDA M to accept it. M has two forks, one of which compares n to m and the other of which compares m to p (skipping over the a's). L1 = {$a^nb^mc^p$: n = m and m = p} is a subset of L. But L1 = $a^nb^nc^n$, which we know is not context free.

4. (1) is fine. (2) is fine if we don't over interpret it. In particular, although both languages are defined in terms of the variable n, the scope of that variable is a single language. So within each individual language definition, the two occurrences of n are correctly interpreted to be occurrences of a single variable, and thus the values must be same both times. However, when we concatenate the two languages, we still have two separate language definitions with separate variables. So the two n's are different. This is the key. It means that we can't assume that, given {a^nb^n}{b^na^n }, we choose the same value of n for the two strings we choose. For example, we could get $a^2b^2b^3a^3$, which is $a^2b^5a^3$, which is clearly not in $\{a^nb^{2n}a^{n\}}$.

5. (a) **(i)** False, since all regular languages are also context free.
(ii) True.
(iii) False. For example a ∈ L, but is not long enough to contain pumpable substrings.
(iv) False.
(v) True, since the context-free languages are closed under reversal.

(b) $S \Rightarrow a\underline{A}s \Rightarrow a\underline{S}SS \Rightarrow aa\underline{S}S \Rightarrow aaa\underline{S} \Rightarrow aaaa\underline{A}S \Rightarrow aaaaba\underline{S} \Rightarrow aaaabaa$.

(c)



(d) M = ({p, q}, {a, b}, {a, b}, p, {q}, Δ), where Δ =
{((p, ε, ε), (q, S)), ((q, a, a), (q, ε)), ((q, b, b), (q, ε)), ((q, ε, S), (q, aAS)), ((q, ε, S), (q, a)), ((q, ε, A), (q, SbA)), ((q, ε, A), (q, SS)), ((q, ε, A), (q, ba)) }

6. The easy way to show that L = {$xx^Ryy^Rzz^R$: x, y, z ∈ {a, b}*} is context free is to recall that we've already shown that {xx^R: x ∈ {a, b}*} is context free, and the context-free languages are closed under concatenation. But we can also do this directly by giving a grammar for L:

$S \rightarrow AAA$
$A \rightarrow aAa$
$A \rightarrow bAb$
$A \rightarrow \varepsilon$

7. (a) L - R is context free. L - R = L ∩ R' (the complement of R). R' is regular (since the regular languages are closed under complement) and the intersection of a context-free language and a regular language is context-free, so L - R is context free.

(b) R - L need not be context free. R - L = R ∩ L'. But L' may not be context free, since the context-free languages are not closed under complement. (The deterministic ones are, but L may not be deterministic.) If we let R = Σ*, then R - L is exactly the complement of L.

8. M_1, which accepts L_1 = ({1, 2}, {a, b}, {a}, Δ, 1, {2}), Δ =
((1, a, ε), (1, a))
((1, b, a), (2, ε))
((1, ε, ε), (2, ε)
((2, b, a), (2, ε)

((2, ε, a), (2, ε)

M_2, which accepts R_1 = ({1, 2, 3, 4}, {a, b}, δ, 1, {2}), δ =

(1, a, 2)
(1, b, 3)
(2, a, 1)
(2, b, 4)
(3, a, 4)
(3, b, 1)
(4, a, 3)
(4, b, 2)

M_3, which accepts $L_1 \cap R_1$ = ({(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2,), (2, 3), (2, 4)}, {a, b}, {a}, Δ, (1,1), {(2,2)}), Δ =

(((1, 1), a, ε), ((1, 2), a))
(((1, 1), b, a), ((2, 3), ε))
(((1, 2), a, ε), ((1, 1), a))
(((1, 2), b, a), ((2, 4), ε))
(((1, 3), a, ε), ((1, 4), a))
(((1, 3), b, a), ((2, 1), ε))
(((1, 4), a, ε), ((1, 3), a))
(((1, 4), b, a), ((2, 2), ε))
(((2, 1), b, a), ((2, 3), ε))
(((2, 2), b, a), ((2, 4), ε))
(((2, 3), b, a), ((2, 1), ε))
(((2, 4), b, a), ((2, 2), ε))
(((1, 1), ε, ε), ((2, 1), ε))
(((1, 2), ε, ε), ((2, 2), ε))
(((1, 3), ε, ε), ((2, 3), ε))
(((1, 4), ε, ε), ((2, 4), ε))
(((2, 1), ε, a), ((2, 1), ε))
(((2, 2), ε, a), ((2, 2), ε))
(((2, 3), ε, a), ((2, 3), ε))
(((2, 4), ε, a), ((2, 4), ε))

CS 341 Homework 17
Turing Machines

1. Let $M = (K, \Sigma, \delta, s, \{h\})$, where

$K = \{q_0, q_1, h\}$,
$\Sigma = \{a, b,$ ❑, ◊$\}$,
$s = q_0$,

and δ is given by the following table,

q	**σ**	**δ(q,σ)**
q_0	a	(q_1, b)
q_0	b	(q_1, a)
q_0	❑	(h, ❑)
q_0	◊	(q_0, →)
q_1	a	(q_0, →)
q_1	b	(q_0, →)
q_1	❑	(q_0, →)
q_1	◊	(q_1, →)

(a) Trace the computation of M starting from the configuration (q_0, ◊$\underline{a}$abbba).

(b) Describe informally what M does when started in q_0 on any square of a tape.

2. Repeat Problem 1 for the machine $M = (K, \Sigma, \delta, s, \{h\})$, where

$K = \{q_0, q_1, q_2, h\}$,
$\Sigma = \{a, b,$ ❑, ◊$\}$,
$s = q_0$,

and δ is given by the following table (the transitions on ◊ are δ(q, ◊) = (q, ◊), and are omitted).

q	**σ**	**δ(q,σ)**
q_0	a	(q_1, ←)
q_0	b	(q_0, →)
q_0	❑	(q_0, →)
q_1	a	(q_1, ←)
q_1	b	(q_2, →)
q_1	❑	(q_1, ←)
q_2	a	(q_2, →)
q_2	b	(q_2, →)
q_2	❑	(h, ❑)

Start from the configuration (q_0, ◊a$\underline{b}$b❑bb❑❑❑aba).

3. Let M be the Turing machine $M = (K, \Sigma, \delta, s, \{h\})$, where

$K = \{q_0, q_1, q_2, h\}$,
$\Sigma = \{a,$ ❑, ◊$\}$,
$s = q_0$,

and δ is given by the following table.
Let $n \geq 0$. Describe carefully what M does when started in the configuration (q_0, ◊❑$a^n\underline{a}$).

q	σ	δ(q,σ)
q_0	a	(q_1, ←)
q_0	❑	(q_0, ❑)
q_0	◊	(q_0, →)
q_1	a	(q_2, ❑)
q_1	❑	(h, ❑)
q_1	◊	(q_1, →)
q_2	a	(q_2, a)
q_2	❑	(q_0, ←)
q_2	◊	(q_2, →)

4. Design and write out in full a Turing machine that scans to the right until it finds two consecutive a's and then halts. The alphabet of the Turing machine should be {a, b, ❑, ◊}.

5. Give a Turing machine (in our abbreviated notation) that takes as input a string w ∈ {a, b}* and squeezes out the a's. Assume that the input configuration is (s, ◊$\underline{❑}$w) and the output configuration is (h, ◊$\underline{❑}$w'), where w' = w with all the a's removed.

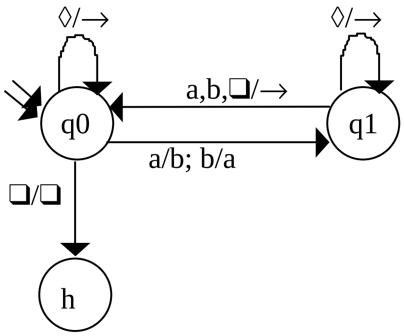
6. Give a Turing machine (in our abbreviated notation) that shifts its input two characters to the right.

Input: ❑w$\underline{❑}$
Output: ❑❑❑w$\underline{❑}$

7. (L & P 5.7.2) Show that if a language is recursively enumerable, then there is a Turing machine that enumerates it without ever repeating an element of the language.

Solutions

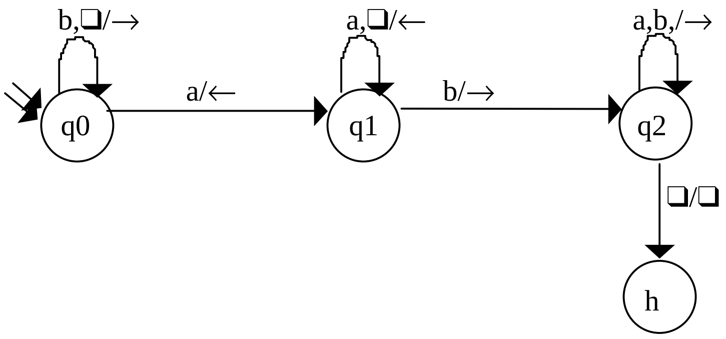
1. (a)



q_0, ◊$\underline{a}$abbba
q_1, ◊$\underline{b}$abbba
q_0, ◊b$\underline{a}$bbba
q_1, ◊b$\underline{b}$bbba
q_0, ◊bb$\underline{b}$bba
q_1, ◊bb$\underline{a}$bba
q_0, ◊bba$\underline{b}$ba
q_1, ◊bba$\underline{a}$ba
q_0, ◊bbaa$\underline{b}$a
q_1, ◊bbaa$\underline{a}$a
q_0, ◊bbaaa$\underline{a}$
q_1, ◊bbaaa$\underline{b}$
q_0, ◊bbaaab$\underline{❑}$
h, ◊bbaaab$\underline{❑}$

(b) Converts all a's to b's, and vice versa, starting with the current symbol and moving right.

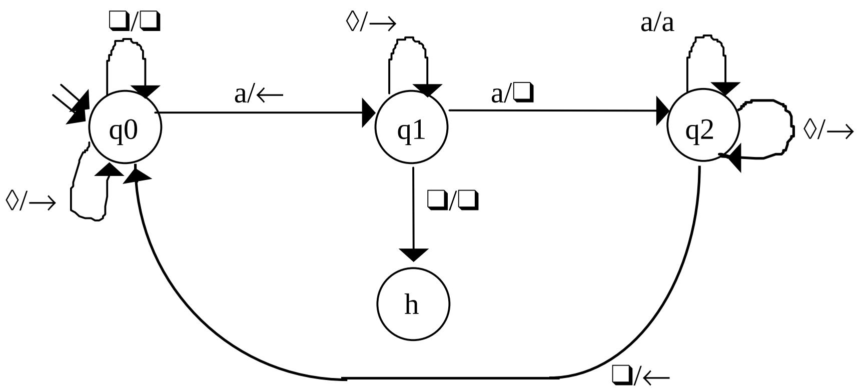
2. (a)



q_0, ◊a$\underline{\text{b}}$b❑bb❑❑❑aba
q_0, ◊ab$\underline{\text{b}}$❑bb❑❑❑aba
q_0, ◊abb$\underline{\text{❑}}$bb❑❑❑aba …
q_0, ◊abb❑bb❑❑❑$\underline{\text{a}}$ba
q_1, ◊abb❑bb❑❑$\underline{\text{❑}}$aba …
q_1, ◊abb❑b$\underline{\text{b}}$❑❑❑aba
q_2, ◊abb❑bb$\underline{\text{❑}}$❑❑aba
h, ◊abb❑bb$\underline{\text{❑}}$❑❑aba

(b) M goes right until if finds an a, then left until it finds a b, then right until it finds a blank.

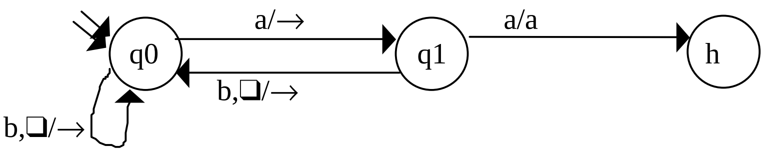
3.



q0, ◊❑a a a a $\underline{\text{a}}$
q1, ◊❑a a a $\underline{\text{a}}$ a
q2, ◊❑a a a $\underline{\text{❑}}$a
q0, ◊❑a a $\underline{\text{a}}$ ❑a
q1, ◊❑a $\underline{\text{a}}$ a ❑a
q2, ◊❑a $\underline{\text{❑}}$a ❑a
q0, ◊❑$\underline{\text{a}}$ ❑a ❑a
q1, ◊$\underline{\text{❑}}$a ❑a ❑a
h, ◊$\underline{\text{❑}}$a ❑a ❑a

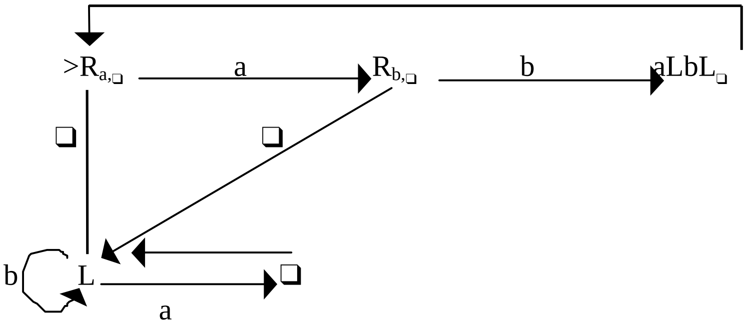
M changes every other a, moving left from the start, to a blank. If n is odd, it loops. If n is even, it halts.

4.

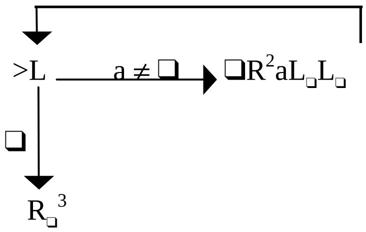


M = (K, Σ, δ, s, {h}), where
K = {q_0, q_1, h},
Σ = {a, b, ❑, ◊},
s = q_0

5. The idea here is that first we'll push all b's to the left, thus squeezing all the a's to the right. Then we'll just replace the a's with blanks. In more detail: scan the string from left to right. Every time we find an a, if there are any b's to the right of it we need to shift them left. So scan right, skipping as many a's as there are. When we find a b, swap it with the last a. That squeezes one a further to the right. Go back to the left end and repeat. Eventually all the a's will come after all the b's. At that point, when we look for a b following an a, all we'll find is a blank. At that point, we just clean up by rewriting all the a's as blanks.



6. The idea is to start with the rightmost character of w, rewrite it as a blank, then move two squares to the right and plunk that character back down. Then scan left for the next leftmost character, do the same thing, and so forth.



7. Suppose that M is the Turing machine that enumerates L. Then construct M* to enumerate L with no repetitions: M* will begin by simulating M. But whenever M outputs a string, M* will first check to see if the string has been output before (see below). If it has, it will just continue looking for strings. If not, it will output it, and it will also write the string, with # in front of it, at the right end of the tape. To check whether a string has been output before, M* just scans its tape checking for the string it is about to output.

CS 341 Homework 18
Computing with Turing Machines

1. Present Turing machines that decide the following languages over {a, b}:

(a) $\varnothing$
(b) $\{\varepsilon\}$
(c) {a}
(d) {a}*

2. Consider the simple (regular, in fact) language $L = \{w \in \{a,b\}^* : |w| \text{ is even}\}$

(a) Give a Turing machine that decides L.

(b) Give a Turing machine that semidecides L.

3. Give a Turing machine (in our abbreviated notation) that accepts $L = \{a^n b^m a^n : m > n\}$

4. Give a Turing machine (in our abbreviated notation) that accepts $L = \{ww : w \in \{a, b\}^*\}$

5. Give a Turing machine (in our abbreviated notation) that computes the following function from strings in {a, b}* to strings in {a, b}* : $f(w) = ww^R$.

6. Give a Turing machine that computes the function f: $\{a,b,c\}^* \rightarrow N$ (the integers), where f(w) = the number of a's (in unary) in w.

7. Let w and x be any two positive integers encoded in unary. Show a Turing machine M that computes

$$f(w, x) = w + x.$$

Represent the input to M as

◊❑w;x❑

8. Two's complement form provides a way to represent both positive and negative binary integers. Suppose that the number of bits allocated to each number is k (generally the word size). Then each positive integer is represented simply as its binary encoding, with leading zeros. Each negative integer n is represented as the result of subtracting |n| from 2^k, where k is the number of bits to be used in the representation. Given a fixed k, it is possible to represent any integer n if $-2^{k-1} \le n \le 2^{k-1} -1$. The high order digit of each number indicates its sign: it is zero for positive integers and 1 for negative integers.
Examples, for k = 4:

0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, 4 = 0100, 5 = 0101, 6 = 0110, 7 = 0111
-1 = 1111, -2 = 1110, -3 = 1101, -4 = 1100, -5 = 1011, -6 = 1010, -7 = 1001, -8 = 1000

Since Turing machines don't have fixed length words, we'd like to be able to represent any integer. We will represent positive integers with a single leading 0. We will represent each negative integer n as the result of subtracting n from 2^{i+1}, where i is the smallest value such that $2^i \ge |n|$. For example, -65 will be represented as 1111111, since 2^7 (128) $\ge$ 65, so we subtract 65 (01000001 in binary) from 2^8 (in binary, 100000000). We need the extra digit (i.e., we subtract from 2^{i+1} rather than from 2^i) because, in order for a positive number to be interpreted as positive, it must have a leading 0, thus consuming an extra digit.

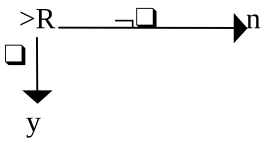
Let w be any integer encoded in two's complement form. Show a Turing machine that computes f(w) = -w.

Solutions

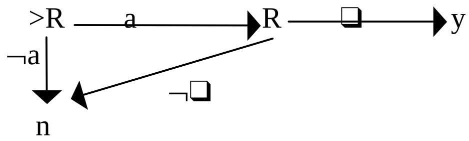
1 (a) We should reject everything, since no strings are in the language.

> n

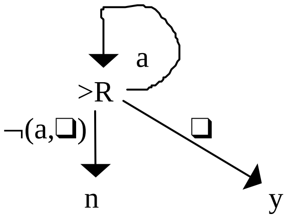
(b) Other than the left boundary symbol, the tape should be blank: ◊❑❑❑



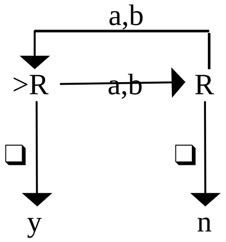
(c) Just the single string a: ◊❑a❑



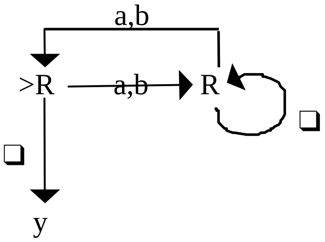
(d) Any number of a's:



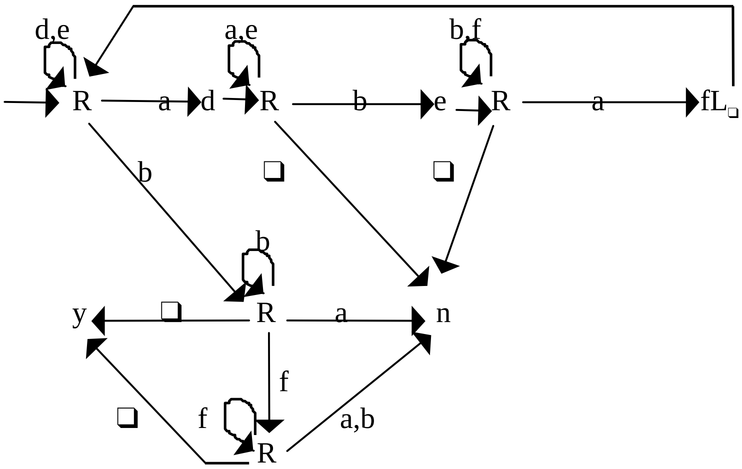
2. (a)



(b)



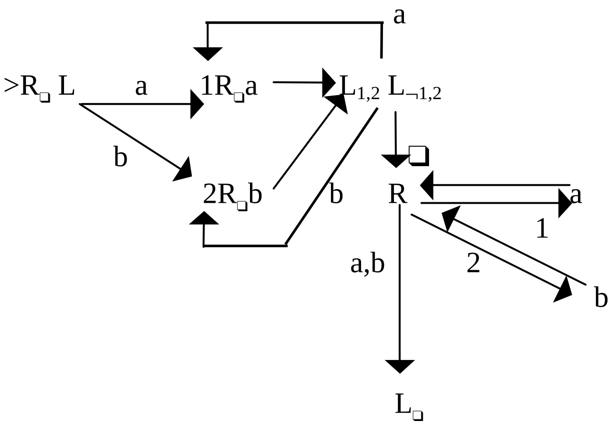
3. The idea is to make a sequence of passes over the input. On each pass, we mark off (with d, e, and f) a matching a, b, and a. This corresponds to the top row of the machine shown here. When there are no matching groups left, then we accept if there is nothing left or if there are b's in the middle. If there is anything else, we reject. It turns out that a great deal of this machine is essentially error checking. We get to the R on the second row as soon as we find the first "extra" b. We can loop in it as long as we find b's. If we find a's we reject. If we find a blank, then the string had just b's, which is okay, so we accept. Once we find an f, we have to go to the separate state R on the third row to skip over the f's and make sure we get to the final blank without either any more b's or any more a's.



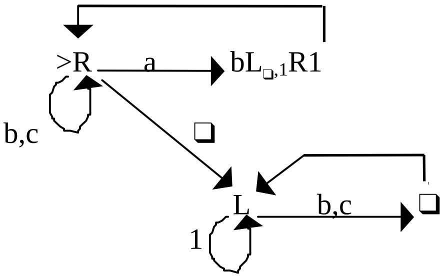
4. The hard part here is that we don't know where the middle of the string is. So we don't know where the boundary between the first occurrence of w ends and the second begins. We can break this problem into three subroutines, which will be executed in order:
(1) Find the middle and mark it. If there's a lone character in the middle (i.e., the length of the input string isn't even), then reject immediately.
(2) Bounce back and forth between the beginning of the first w and the beginning of the second, marking off characters if they match and rejecting if they don't.
(3) If we get to the end of the w's and everything has matched, accept.

Let's say a little more about step (1). We need to put a marker in the middle of the string. The easiest thing to do is to make a double marker. We'll use ##. That way, we can start at both ends (bouncing back and forth), moving a marker one character toward the middle at each step. For example, if we start with the tape ◊❑aabbaabb❑❑❑, after one mark off step we'll have ◊❑a#abbaab#b❑❑❑, then ◊❑aa#bbaa#bb❑❑❑, and finally ◊❑aabb##aabb❑❑❑. So first we shift the whole input string two squares to the right on the tape to make room for the two markers. Then we bounce back and forth, moving the two markers toward the center. If they meet, we've got an even length string and we can continue. If they don't, we reject right away.

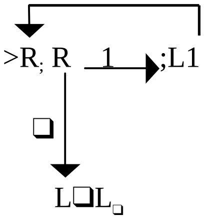
5. The idea is to work from the middle. We'll scan right to the rightmost character of w (which we find by scanning for the first blank, then backing up (left) one square. We'll rewrite it so we know not to deal with it again (a's will become 1's; b's will become 2's.) Then we move right and copy it. Now if we scan back left past any 1's or 2's, we'll find the next rightmost character of w. We rewrite it to a 1 or a 2, then scan right to a blank and copy it. We keep this up until, when we scan back to the left, past any 1's or 2's, we hit a blank. That means we've copied everything. Now we scan to the right, replacing 1's by a's and 2's by b's. Finally, we scan back to the left to position the read head to the left of w.



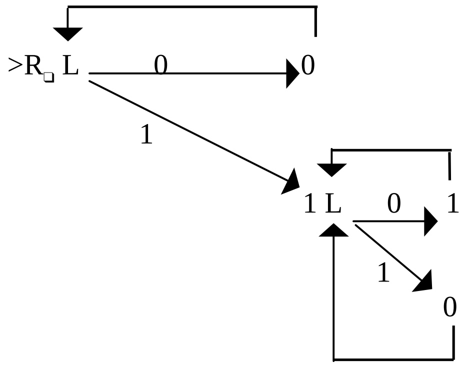
6. The idea here is that we need to write a 1 for every a and we can throw away b's and c's. We want the 1's to end up at the left end of the string, so then all we have to do to clean up at the end is erase all the b's and c's to the right of the area with the 1's. So, we'll start at the left edge of the string. We'll skip over b's and c's until we get to an a. At that point, rewrite it as b so we don't count it again. Then (remembering it in the state) scan left until we get to the blank (if this is the first 1) or we get to a 1. In either case, move one square to the right and write a 1. We are thus overwriting a b or a c, but we don't care. We're going to throw them away anyway. Now start again scanning to the right looking for an a. At some point, we'll come to the end of string blank instead. At that point, just travel leftward, rewriting all the b's and c's to blank, then cross the 1's and land on the blank at the left of the string.



7. All we have to do is to concatenate the two strings. So shift the second one left one square, covering up the semicolon.



8. Do a couple of examples of the conversion to see what's going on. What you'll observe is that we want to scan from the right. Initially, we may see some zeros, and those will stay as zeros. If we ever see a 1, then we rewrite the first one as a 1. After that, we're dealing with borrowing, so we swap all digits: every zero becomes a one and every one becomes a zero, until we hit the blank at the end of the string and halt.



CS 341 Homework 19
Turing Machine Extensions

1. Consider the language L = {ww^R}.
(a) Describe a one tape Turing machine to accept L.

(b) Describe a two tape Turing machine to accept L.

(c) How much more efficient is the two tape machine?

2. Give (in abbreviated notation) a nondeterministic Turing machine that accepts the language

$$L = \{ww^Ruu^R : w, u \in \{a, b\}^*\}$$

Solutions

(1) (a) The one tape machine needs to bounce back and forth between the beginning of the input string and the end, marking off matching symbols.

(b) The two tape machine works as follows: If the input is ε, accept. If not, copy the input to the second tape and record in the state that you have processed an even number of characters so far. Now, start the first tape at the left end and the second tape at the right end . Check that the symbols on the two tapes are the same. If not, reject. If so, move the first tape head to the right and the second tape head to the left. Also record that you have processed an odd number and continue, each time using the state to keep track of whether you've seen an even or odd number of characters so far. When you reach the end of the input tape, accept if you've seen an even number of characters. Reject if you've seen an odd number. (The even/odd counter is necessary to make sure that you reject strings such as aba.)

(c) The one tape machine takes time proportional to the square of the length of the input, since for an input of length n it will make n passes over the input, each of which takes on average n/2 steps. The two tape machine takes time that's linear in n. It takes n steps to copy, then another n steps to compare.

2. The idea is just to use nondeterminism to guess the location of the boundary between the w and u regions. Each path will choose a spot, shift the u region to the right, and insert a boundary marker #. Once this is done, the machine simply checks each region for ww^R. If we get a string in L, one of the guessed paths will work.

CS 341 Homework 20
Unrestricted Grammars

1. Find grammars that generate the following languages:
(a) L = {ww : w $\in$ {a, b}*}
(b) L = { a^{2^n} : n $\geq$ 0}
(c) L = { $a^nb^{2n}c^{3n}$: n $\geq$ 1}
(d) L = {w^R : w is the social security number of a living American citizen}
(e) L = {wc^md^n : w $\in$ {a, b}* and m = the number of a's in w and n equals the number of b's in w}

2. Find a grammar that computes the function f(w) = ww, where w $\in$ {a, b}*.

Solutions

1. (a) L = {ww : w $\in$ {a, b}*}
There isn't any way to generate the two w's in the correct order. Suppose we try. Then we could get aSa. Suppose we want b next. Then we need Sa to become bSab, since the new b has to come after the a that's already there. That could work. Now we have abSab. Let's say we want a next. Now Sab has to become aSaba. The problem is that, as the length of the string grows, so does the number of rules we'll need to cope with all the patterns we could have to replace. In a finite number of rules, we can't deal with replacing S (which we need to do to get the next character in the first occurrence of w), and adding a new character that is arbitrarily far away from S.

The other approach we could try would be have a rule S $\rightarrow$ WW, and then let W generate a string of a's and b's. But this won't work, since we have no way to control the expansion of the two W's so that they produce the same thing.

So what we need to do is to generate ww^R and then, carefully, reverse the order of the characters in w^R. What we'll do is to start by erecting a wall (#) at the right end of the string. Then we'll generate ww^R. Then, in a second phase, we'll take the characters in the second w and, one at a time, starting with the leftmost, move it right and then move it past the wall. At each step, we move each character up to the wall and then just over it, but we don't reverse characters once they get over the wall. The first part of the grammar, which will generate wTw^R, looks like this:

S $\rightarrow$ S_1 #	This inserts the wall at the right.
$S_1 \rightarrow aS_1a$	
$S_1 \rightarrow bS_1b$	
$S_1 \rightarrow T$	T will mark the left edge of the portion that needs to be reversed.

At this point, we can generate strings such as abbbTbbba#. What we need to do now is to reverse the string of a's and b's that is between T and #. To do that, we let T spin off a marker Q, which we can pass rightward through the string. As it moves to the right, it will take the first a or b it finds with it. It does this by swapping the character it is carrying (the one just to the right of it) with the next one to the right. It also moves itself one square to the right. The four rules marked with * accomplish this. When Q's character gets to the # (the rules marked **), the a or b will swap places with the # (thus hopping the fence) and the Q will go away. We can keep doing this until all the a's and b's are behind the fence and in the right order. Then the final T# will drop out. Here are the rules for this phase:

$T \rightarrow TQ$
$Qaa \rightarrow aQa$ *
$Qab \rightarrow bQa$ *
$Qbb \rightarrow bQb$ *
$Qba \rightarrow aQb$ *
$Qa\# \rightarrow \#a$ **
$Qb\# \rightarrow \#b$ **
$T\# \rightarrow \varepsilon$
So with R as given above, the grammar G = ({S, S_1, #, T, Q, a, b}, {a, b}, R, S}

(b) L = { a^{2^n} : n ≥ 0}
The idea here is first to generate the first string, which is just a. Then think about the next one. You can derive it by taking the previous one, and, for every a, write two a’s. So we get aa. Now to get the third one, we do the same thing. Each of the two a’s becomes two and we have four, and so forth. So we need a rule to get us started and to indicate the possibility of duplication. Then we need rules to actually do the duplication. To make duplication happen, we need a symbol that gets generated by S indicating the option to repeat. We’ll use P. Since duplication can happen an arbitrary number of times, we need P to spin off as many individual duplication commands as we want. We’ll use R for that. The one other thing we need is to make sure, if we start a duplication step, that we finish it. In other words, suppose we currently have aaaa. If we start duplicating the a’s, we must duplicate all of them. Otherwise, we might end up with, for example, seven a’s. So we’ll introduce a left edge marker, #. Once we fire up a duplication (by creating an R), we’ll only stop (i.e., get rid of R) when R has made it all the way to the other end of the string (namely the left end since it starts at the right). So we get the following rules:

$S \rightarrow \#aP$ P lets us start up duplication processes as often as we like.
$P \rightarrow \varepsilon$ When we’ve done as many as we want, we get rid of P.
$P \rightarrow RP$ R will actually do a duplication by moving leftward, duplicating every a it sees.
$aR \rightarrow Raa$ Actually duplicates one a, and moves R one square to the left so it moves on to the next a
$\#R \rightarrow \#$ Get rid of R once it's made it all the way to the left
$\# \rightarrow \varepsilon$ Get of # at the end
So with R as given above, the grammar G = ({S, P, R, #, a, b}, {a, b}, R, S}

(c) L = { $a^n b^{2n} c^{3n}$: n ≥ 1}
This one is very similar to $a^n b^n c^n$. The only difference is that we will churn out b's in pairs and c's in triples each time we expand S. So we get:
$S \rightarrow aBSccc$
$S \rightarrow aBccc$
$Ba \rightarrow aB$
$Bc \rightarrow bbc$
$Bb \rightarrow bbb$
So with R as given above, the grammar G = ({S, B, a, b, c}, {a, b, c}, R, S}

(d) L = {w^R : w is the social security number of a living American citizen}
This one is regular. There is a finite number of such social security numbers. So we need one rule for each number. Each rule is of the form S → <valid number>. So with that collection of rules as R, the grammar G = ({S, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, R, S}

(e) L = {$wc^m d^n$: w ∈ {a, b}* and m = the number of a's in w and n equals the number of b's in w}
The idea here is to generate a c every time we generate an a and to generate a d every time we generate a b. We'll do this by generating the nonterminals C and D, which we will use to generate c's and d's once everything is in the right place. Once we've finished generating all the a's and b's we want, the next thing we need to do is to get

all the D's to the far right of the string, all the C's next, and then have the a's and b's left alone at the left. We guarantee that everything must line up that way by making sure that C can't become c and D can't become d unless things are right. To do this, we require that D can only become d if it's all the way to the right (i.e., it's followed by #) or it's got a d to its right. Similarly with C. We can do this with the following rules;

$S \rightarrow S_1\#$
$S_1 \rightarrow aS_1C$
$S_1 \rightarrow bS_1D$
$S_1 \rightarrow \varepsilon$
$DC \rightarrow CD$
$D\# \rightarrow d$
$Dd \rightarrow dd$
$C\# \rightarrow c$
$Cd \rightarrow cd$
$Cc \rightarrow cc$
$\# \rightarrow \varepsilon$
So with R as given above, the grammar G = ({S, S_1, C, D, #, a, b, c, d}, {a, b, c, d}, R, S}

2. We need to find a grammar that computes the function f(w) = ww. So we'll get inputs such as SabaS. Think of the grammar we'll build as a procedure, which will work as described below. At any given time, the string that has just been derived will be composed of the following regions:

<the part of w that has already been copied>	S	<the part of w that has not yet been copied, which may have within it a character (preceded by #) that is currently being copied by being moved through the region>	T (inserted when the first character moves into the copy region)	<the part of the second w that has been copied so far, which may have within it a character (preceded by %) that is currently being moved through the region>	W (also inserted when T is)

Most of the rules come in pairs, one dealing with an a, the other with b.

$SS \rightarrow \varepsilon$	Handles the empty string.
$Sa \rightarrow aS\#a$	Move S past the first a to indicate that it has already been copied. Then start copying it by introducing a new a, preceded by the special marker #, which we'll use to push the new a to the right end of the string.
$Sb \rightarrow bS\#b$	Same for copying b.
$\#aa \rightarrow a\#a$	Move the a we're copying past the next character if it's an a.
$\#ab \rightarrow b\#a$	Move the a we're copying past the next character if it's a b.
$\#ba \rightarrow a\#b$	Same two rules for pushing b.
$\#bb \rightarrow b\#b$	"
$\#aS \rightarrow \#aTW$	We've gotten to the end of w. This is the first character to be copied, so the initial S is at the end of w. We need to create a boundary between w and the copied w. T will be that boundary. We also need to create a boundary for the end of the copied w. W will be that boundary. T and W are adjacent at this point because we haven't copied any characters into the copy region yet.
$\#bS \rightarrow \#aTW$	Same if we get to the end of w pushing b.
$\#aT \rightarrow T\%a$	Jump the a we're copying into the copy region (i.e., to the right of T). Get rid of #, since we're done with it. Introduce %, which we'll use to push the copied a through the copy region.
$\#bT \rightarrow T\%b$	Same if we're pushing b.

%aa → a%a	Push a to the right through the copied region in exactly the same way we pushed it through w, except we're using % rather than # as the pusher. This rule pushes a past a.
%ab → b%a	Pushes a past b.
%ba → a%b	Same two rules for pushing b.
%bb → b%b	"
%aW → aW	We've pushed an a all the way to the right boundary, so get rid of %, the pusher.
%bW → bW	Same for a pushed b.
ST → ε	All the characters from w have been copied, so they're all to the left of S, which causes S to be adjacent to the middle marker T. We can now get rid of our special walls. Here we get rid of S and T.
W → ε	Gid rid of W. Note that if we do this before we should, there's no way to get rid of %, so any derivation path that does this will fail to produce a string in {a, b}*.

So with R as given above, the grammar G = ({S, T, W, #, %,a, b}, {a, b}, R, S}

CS 341 Homework 21
Undecidability

1. Which of the following problems about Turing machines are solvable, and which are undecidable? Explain your answers carefully.
(a) To determine, given a Turing machine M, a state q, and a string w, whether M ever reaches state q when started with input w from its initial state.
(b) To determine, given a Turing machine M and a string w, whether M ever moves its head to the left when started with input w.
(c) To determine, given two Turing machines, whether one semidecides the complement of the language semidecided by the other.
(d) To determine, given a Turing machine M, whether the language semidecided by M is finite.

2. Show that it is decidable, given a pushdown automaton M with one state, whether L(M) = Σ*. (Hint: Show that such an automaton accepts all strings if and only if it accepts all strings of length one.)

3. Which of the following problems about context-free grammars are solvable, and which are undecidable? Explain your answers carefully.
(a) To determine, given a context-free grammar G, is ε ∈ L(G)?
(b) To determine, given a context-free grammar G, is {ε} = L(G)?
(c) To determine, given two context-free grammars G_1 and G_2, is $L(G_1) \subseteq L(G_2)$?

4. The nonrecursive languages L that we have discussed in class all have the property that either L or the complement of L is recursively enumerable.
(a) Show by a counting argument that there is a language L such that neither L nor its complement is recursively enumerable.
(b) Give an example of such a language.

Solutions

1. (a) To determine, given a Turing machine M, a state q, and a string w, whether M ever reaches state q when started with input w from its initial state. This is not solvable. We can reduce H to it. Essentially, if we can tell whether a machine M ever reaches some state q, then let q be M's halt state (and we can massage M so it has only one halt state). If it ever gets to q, it must have halted. More formally:

L_1 = H = {s = "M" "w" : M halts on input string w}

⇓ τ

(?M_2) L_2 = {s : "M" "w" "q" : M reaches state q when started with input w from its initial state}

Let τ' create, from M the machine M* as follows. Initially M* equals M. Next, a new halting state H is created in M*. Then, from each state that was a halting state in M, we create transitions in M* such that for all possible values of the current tape square, M* goes to H. We create no other transitions to H. Notice that M* will end up in H in precisely the same situations in which M halts.

Now let τ("M" "w") = τ'("M") "w" "H"

So, if M_2 exists, then M_1 exists. It invokes τ' to create M*. Then it passes "M*", "w", and "H" to M_2 and returns whatever M_2 returns. But M_1 doesn't exist. So neither does M_2.

(b) To determine, given a Turing machine M and a string w, whether M ever moves its head to the left when started with input w. This one is solvable. We will assume that M is deterministic. We can build the deciding machine D as follows. D starts by simulating the operation of M on w. D keeps track on another tape of each configuration of M that it has seen so far. Eventually, one of the following things must happen:

1. M moves its head to the left. In this case, we say yes.
2. M is stuck on some square s of the tape. In other words, it is in some state p looking at some square s on the tape and it has been in this configuration before. If this happens and M didn't go left yet, then M simply hasn't moved off of s. And it won't from now on, since it's just going to do the same thing at this point as it did the last time it was in this configuration. So we say no.
3. M moves off the right hand edge of the input w. So it is in some state p looking at a blank. Within k steps (if k is the number of states in M), M must repeat some state p. If it does this without moving left, then again we know that it never will. In other words, if the last time it was in the configuration in which it was in state p, looking at a blank, there was nothing to the right except blanks, and it can't move left, and it is again in that same situation, it will do exactly the same thing again. So we say no.

(c) To determine, given two Turing machines, whether one semidecides the complement of the language semidecided by the other. This one is not solvable. We can reduce to it the problem, "Given a Turing machine M, is there any string at all on which M halts?" (Which is equivalent to "Is $L(M) = \varnothing$?") In the book we show that this problem is not solvable. What we'll do is to build a machine M* that semidecides the language Σ^*, which is the complement of the language $\varnothing$. If we could build a machine to tell, given two Turing machines, whether one semidecides the complement of the language semidecided by the other, then to find out whether any given machine M accepts anything, we'd pass M and our constructed M* to this new machine. If it says yes, then M accepts $\varnothing$. If it says no, then M must accept something. Formally:

$L_1 =$ {s = "M" M halts on some string w}

$\Downarrow \quad \tau$

(?M_2) $L_2 =$ {s = "M_1" "M_2" : M_1 decides the complement of the language semidecided by M_2}

M accepts strings over some input alphabet Σ. Let τ' construct a machine M* that semidecides the language Σ^*. Then τ("M") = "M" "τ'(M)".

So, if M_2 exists, then M_1 exists. It invokes τ' to create M*. Then it passes "M" and "M*" to M_2 and returns the opposite of whatever M_2 returns (since M2 says yes if $L(M) = \varnothing$ and M1 wants to say yes if $L(M) \neq \varnothing$). But M_1 doesn't exist. So neither does M_2.

(d) To determine, given a Turing machine M, whether the language semidecided by M is finite. This one isn't solvable. We can reduce to it the problem, "Given a Turing machine M, does M halt on ε?" We'll construct, from M, a new machine M*, which erases its input tape and then simulates M. M* halts on all inputs iff M halts on ε. If M doesn't halt on ε, then M* halts on no inputs. So there are two situations: M* halts on all inputs (i.e., L(M*) is infinite) or M* halts on no inputs (i.e., L(M*) is finite). So, if we could build a Turing machine M_2 to decide whether L(M*) is finite or infinite, we could build a machine M_1 to decide whether M halts on ε. Formally:

$L_1 =$ {s = "M" M halts on ε}

$\Downarrow \quad \tau$

(?M_2) $L_2 =$ {s = "M" is finite}

Let τ construct the machine M* from "M" as described above.

So, if M_2 exists, then M_1 exists. It invokes τ to create M* which accepts a finite language precisely if M accepts ε. But M_1 doesn't exist. So neither does M_2.

2. M only has one state S. If S is not a final state, then L(M) = ∅, which is clearly not equal to Σ*, so we say no. Now suppose that S is a final state. Then M accepts ε. Does it also accept anything else? To accept any single character c in Σ, there must be a transition ((S, c, ε), (S, ε)). In other words, we must be able to end up in S with an empty stack if, looking at an empty stack, we see c. If there is not such a transition for every element c of Σ, then we say no, since we clearly cannot get even all the one character strings in Σ*. Now, suppose that all those required transitions do exist. Then, we can stay in S with an empty stack (and thus accept) no matter what character we see next and no matter what is on the stack (since these transitions don't check the stack). So, if M accepts all strings in Σ* of length one, then it accepts all strings in Σ*. Note that if M is deterministic, then if it does have all the required transitions it will have no others, since all possible configurations are accounted for

3. (a) To determine, given a context-free grammar G, is ε ∈ L(G) This is solvable by using either top down or bottom up parsing on the string ε.

(b) To determine, given a context-free grammar G, is {ε} = L(G) This is solvable. By the context-free pumping theorem, we know that, given a context-free grammar G generating a language L(G), if there is a string of length greater than B^T in L, then vy can be pumped out to create a shorter string also in L (the string must be shorter since |vy| >0). We can, of course, repeat this process until we reduce the original string to one of length less than B^T. This means that if there any strings in L, there are some strings of length less than B^T. So, to see whether L = {ε}, we do the following: First see whether ε ∈ L(G) by parsing. If not, we say no. If ε is in L, then we need to determine whether any other strings are also in L. To do this, we test all strings in Σ* of length up to B^{T+1}. If we find one, we say no, L ≠ {ε}. If we don't find any, we can assert that L = {ε}. Why? If there is a longer string in L and we haven't found it yet, then we know, by the pumping theorem, that we could pump out vy until we got a string of length B^T or less. If ε were not in L, we could just test up to length B^T and if we didn't find any elements of L at all, we could stop, since if there were bigger ones we could pump out and get shorter ones but there aren't any. However, because ε is in L, what about the case where we pump out and get ε? That's why we go up to B^{T+1}. If there are any long strings that pump out to ε, then there is a shortest such string, which can't be longer than B^{T+1} since that's the longest string we can pump out (by the strong version of the pumping theorem).

(c) To determine, given two context-free grammars G_1 and G_2, is $L(G_1) \subseteq L(G_2)$ This isn't solvable. If it were, then we could reduce the unsolvable problem of determining whether $L(G_1) = L(G_2)$ to it. Notice that $L(G_1) = L(G_2)$ iff $L(G_1) \subseteq L(G_2)$ and $L(G_2) \subseteq L(G_1)$. So, if we could solve the subset problem, then to find out whether $L(G_1) = L(G_2)$, all we do is ask whether the first language is a subset of the second and vice versa. If both answers are yes, we say yes. Otherwise, we say no. Formally:

L_1 = {s : s = G_1 G_2, G_1 and G_2 are context-free grammars, and $L(G_1) = L(G_2)$ }

⇓ τ

(?M_2) L_2 = {s : s = G_1 G_2, G_1 and G_2 are context-free grammars, and $L(G_1) \subseteq L(G_2)$ }

If M_2 exists, then $M_1(G_1\ G_2) = M_2(G_1\ G_2)$ AND $M_2(G_2\ G_1)$. To write this out in our usual notation so that the last function that gets applied is M_2, is sort of tricky, but it can, of course be done: Don't worry about doing it. If you can write any function for M_1 that is guaranteed to be recursive if M_2 exists, then you've done the proof.

4. (a) If any language L is recursively enumerable, then there is a Turing machine that semidecides it. Every Turing machine has a description of finite length. Therefore, the number of Turing machines, and thus the number of recursively enumerable languages, is countably infinite (since the power set of a countable set is countably infinite). If, for some language L, its complement is re, then it must have a semideciding Turing machine, so there is a countably infinite number of languages whose complement is recursively enumerable. But there is an uncountable number of languages. So there must be languages that are not recursively enumerable and do not have recursively enumerable complements.

(b) L = {"M" : M halts on the input 0 and M doesn't halt on the input 1}.
The complement of L = {"M" : M doesn't halt on the input 0 or M halts on the input 1}. Neither of these languages is recursively enumerable because of the doesn't halt piece.

CS 341 Homework 22
Review

1. Given the following language categories:

A: L is finite.
B: L is not finite but is regular.
C: L is not regular but is deterministic context free
D: L is not deterministic context free but is context free
E: L is not context free but is Turing decidable
F: L is not Turing decidable but is Turing acceptable
G: L is not Turing acceptable

Assign the appropriate category to each of the following languages. Make sure you can justify your answer.

a. _____ $\{a^nb^{kn} : k = 1 \text{ or } k = 2, n \geq 0\}$
b. _____ $\{a^nb^{kn} : k = 0 \text{ or } k = 1, n \geq 0\}$
c. _____ $\{a^nb^nc^n : n \geq 0\}$
d. _____ $\{a^nb^nc^m : n \geq 0, m \geq 0\}$
e. _____ $\{a^nb^n : n \geq 0\} \cup a^*$
f. _____ $\{a^nb^m :$ n is prime and m is even$\}$
g. _____ $\{a^nb^mc^{m+n} : n \geq 0, m \geq 0\}$
h. _____ $\{a^nb^mc^{mn} : n \geq 0, m \geq 0\}$
i. _____ $\{a^nb^m : n \geq 0, m \geq 0\}$
j. _____ $\{xy : x \in a^*, y \in b^*, |x| = |y|\}$
k. _____ $\{xy : x \in a^*, y \in a^*, |x| = |y|\}$
l. _____ $\{x : x \in \{a, b, c\}^*,$ and x has 5 or more a's$\}$
m._____ {"M" : M accepts at least 1 string}
n._____ {"M" : M is a Turing machine that halts on input ε and $|\text{"M"}| \leq 1000$}
o._____ {"M" : M is a Turing machine with ≤ 50 states}
p._____ {"M" : M is a Turing machine such that $L(M) = a^*$}
q. _____ $\{x : x \in \{A, B, C, D, E, F, G\},$ and x is the answer you write to this question$\}$

Solutions

a. __D__ $\{a^nb^{kn} : k = 1 \text{ or } k = 2, n \geq 0\}$

We haven't discussed many techniques for proving that a context free language isn't deterministic, so we can't prove that this one isn't. But essentially the reason this one isn't is that we don't know what to do when we see b's. Clearly, we can build a pda M to accept this language. As M reads each a, it pushes it onto the stack. When it starts seeing b's, it needs to start popping a's. But there's no way to know, until either it runs out of b's or it gets to the n+1st b, whether to pop an a for each b or hold back and pop an a for every other b. So M is not deterministic.

b. __C__ $\{a^nb^{kn} : k = 0 \text{ or } k = 1, n \geq 0\}$

This one is looks very similar to **a**, but it's different in one key way. Remember that the definition of deterministic context free is that it is possible to build a deterministic pda to accept L$. So now, we can build a deterministic pda M as follows: Push each a onto the stack. When we run out of a's, the next character will either be $ (in the case where k = 0) or b (in the case where k = 1). So we know right away which case we're dealing with. If M sees a b, it goes to a state where it pops one b for each a and accepts if it comes out even. If it sees $, it goes to a state where it clears the stack and accepts.

c. __E__ $\{a^nb^nc^n : n \geq 0\}$

We proved that this is recursive by showing a grammar for it in Lecture Notes 24. We used the pumping theorem to prove that it isn't context free in Lecture Notes 19.

d. __C__ $\{a^nb^nc^m : n \geq 0, m \geq 0\}$
This one is context free. We need to compare the a’s to the b’s, but the c’s are independent. So a grammar to generate this one is:

S → A C
A → a A b
A → ε
C → c C
C → ε

It’s deterministic because we can build a pda that always knows what to do: push a’s, pop an a for each b, then simply scan the c’s.

e. __C__ $\{a^nb^n : n \geq 0\} \cup a^*$
This one is equivalent to **b**, since $a^* = a^nb^{0n}$.

f. __E__ $\{a^nb^m : n$ is prime and m is even$\}$
This one is recursive because we can write an algorithm to determine whether a number is prime and another one to determine whether a number is even. The proof that it is essentially the same as the one we did in class that a^n: n is prime is not context free.

g. __C__ $\{a^nb^mc^{m+n} : n \geq 0, m \geq 0\}$
This one is context free. A grammar for it is:

S → a S c
S → b S c
S → ε

It’s deterministic because we can build a deterministic pda M for it: M pushes each a onto its stack. It also pushes an a for each b. Then, when it starts seeing c’s, it pops one a for each c. If it runs out of a’s and c’s at the same time, it accepts.

h. __E__ $\{a^nb^mc^{mn} : n \geq 0, m \geq 0\}$
This one is similar to **g**, but because the number of c’s is equal to the product of n and m, rather than the sum, there is no way to know how many c’s to generate until we know both how many a’s there are and how many b’s. Clearly we can write an algorithm to do it, so it’s recursive. To prove this , we need to use the pumping theorem. Let $w = a^Mb^Mc^{MM}$. Call the a’s, region 1, the b’s region 2, and the c’s region 3. Clearly neither v nor y can span regions since, if they did, we’d get a string with letters out of order. So we need only consider the following possibilities:

(1, 1) The number of c’s will no longer be the product of n and m.
(1, 2) The number of c’s will no longer be the product of n and m.
(1, 3) Ruled out by $|vxy| \leq M$.
(2, 2) The number of c’s will no longer be the product of n and m.
(2, 3) The number of c’s will no longer be the product of n and m.
(3, 3) The number of c’s will no longer be the product of n and m.

i. __B__ $\{a^nb^m : n \geq 0, m \geq 0\}$
This one is regular. It is defined by the regular expression a*b*. It isn’t finite, which we know from the presence of Kleene star in the regular expression.

j. __C__ $\{xy : x \in a^*, y \in b^*, |x| = |y|\}$
This one is equivalent to a^nb^n, which we’ve already shown is context free and not regular. W showed a deterministic pda to accept it in Lecture Notes 14.

k. __B__ $\{xy : x \in a^*, y \in a^*, |x| = |y|\}$
This one is $\{w = a^*: |w|$ is even$\}$. We’ve shown a simple two state FSM for this one.

l. __B__ $\{x : x \in \{a, b, c\}^*,$ and x has 5 or more a's$\}$
This one also has a simple FSM F that accepts it. F has six states. It simply counts a’s, up to five. If it ever gets to 5, it accepts.

m.__F__ {"M" : M accepts at least 1 string}
This one isn't recursive. We know from Rice's Theorem that it can't be, since another way to say this is
{"M" : L(M) contains at least 1 string}
We can also show that this one isn't recursive by reduction, which is done in the Supplementary Materials.

n.__A__ {"M" : M is a Turing machine that halts on input ε and |"M"| ≤ 1000}
This one is finite because of the limit on the length of the strings that can be used to describe M. So it's finite (and thus regular) completely independently of the requirement that M must halt on ε. You may wonder whether we can actually build a finite state machine F to accept this language. What we know for sure is that F exists. It must for any finite language. Whether we can build it or not is a separate question. The undecidability of the halting problem tells us that we can't build an algorithm to determine whether an arbitrary TM M halts on ε. But that doesn't mean that we can't look at most Turing Machines and tell. So, here, it is likely that we could write out all the TMs of length less than 1000 and figure out which ones accept ε. We could then build a deciding FSM F. But even if we can't, that doesn't mean that no such FSM exists. It just means that we don't know what it is. This is no different from the problem of building an FSM to accept all strings of the form mm/dd/yy, such that mm/dd/yy is your birthday. A simple machine F to do this exists. You know how to write it. I don't because I don't know when your birthday is. But that fact that I don't know how to build F says nothing about its existence.

o.__E__ {"M" : M is a Turing machine with ≤ 50 states}
This one looks somewhat similar to **n**. But it's different in a key way. This set isn't finite because there is no limit on the number of tape symbols that M can use. So we can't do the same trick we can do in **n**, where we could simply list all the machines that met the length restriction. With even a single state, I can build a TM whose description is arbitrarily long. I simply tell it what to do in state one if it's reading character 1. Then what to do if it's reading character 2. Then character 3, and so forth. There's no limit to the number of characters, so there's no limit to the length of the string I must write to consider all of them. Given that the language is not finite, we need a TM to decide it. Why? What we need to do is to check to make sure that the string is a syntactically valid encoding of a Turing Machine. Recall the syntax of an encoding. When we see the first a??? symbol that encodes a tape symbol, we know how many digits it has. All the others must have the same number of digits. So we have to remember that number. Since there's no limit to it, we can't remember it in a finite number of states. Since we need to keep referring to it, we can't remember it on a stack. So we need a TM. But the TM is a straightforward program that will always halt. Thus the language is recursive.

p.__G__ {"M" : M is a Turing machine such that L(M) = a*}
This one isn't recursive. Again, we know that from Rice's Theorem. And we can prove it by reduction, which we did in the supplementary materials for the more general case of any alphabet Σ. But this language is even harder than many we have considered, such as H. It isn't even recursively enumerable. Why? Informally, the TM languages that are re are the ones where we can discover positive instances by simulation (like, H, where we ask whether M halts on a particular w?). But how can we try all strings in a*? Proving this formally is beyond the scope of this class.

q. __A__ {x : x ∈ {A, B, C, D, E, F, G}, and x is the answer you write to this question}
This one is finite. In fact, it is a language of cardinality 1. Thus it's regular and there exists an FSM F that accepts it. You may feel that there's some sort of circularity here. There really isn't, but even if there were, we can use the same argument here that we used in **n**. Even if we didn't know how to build F, we still know that it exists.