University of California, San Diego
Department of Computer Science and Engineering
Course: CSE 105 – Theory of Computability
2004 Summer Session I
July 9, 2004

Instructor: Adriana Palacio
TAs: Ragesh Jaiswal
Vadim Lyubashevsky
Saurabh Panjwani

Problem Set 1 Solutions

1. Give the formal description of the machine pictured in Exercise 1.16 (a) of the textbook.

The formal description is $M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{q_1, q_2\}$
- $\Sigma = \{a, b\}$
- $\delta : Q \times \Sigma \to Q$ is defined as
  $\delta(q_1, a) = q_1$
  $\delta(q_1, b) = q_2$
  $\delta(q_2, a) = q_2$
  $\delta(q_2, b) = q_1$
- $q_0 = q_1$
- $F = \{q_2\}$

---

2. Exercise 1.4 d, j, l.

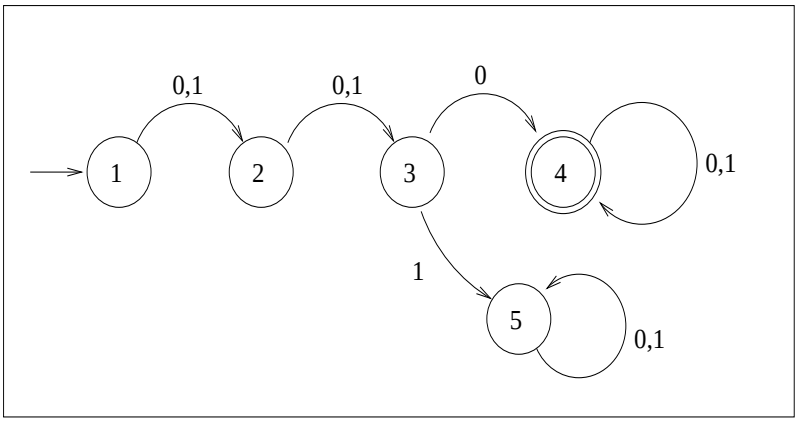


Figure: DFA for exercise 1.4 d

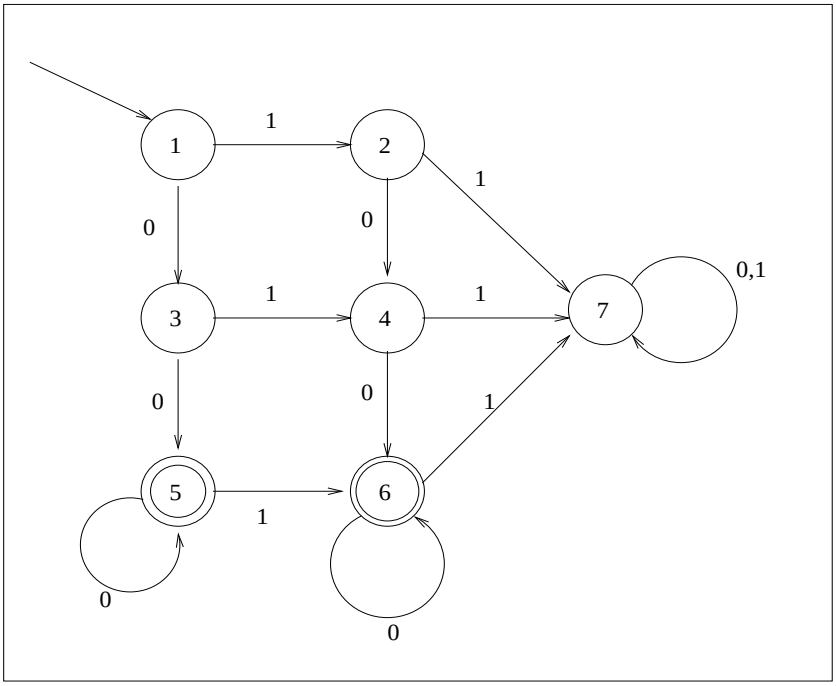


Figure: DFA for exercise 1.4 j

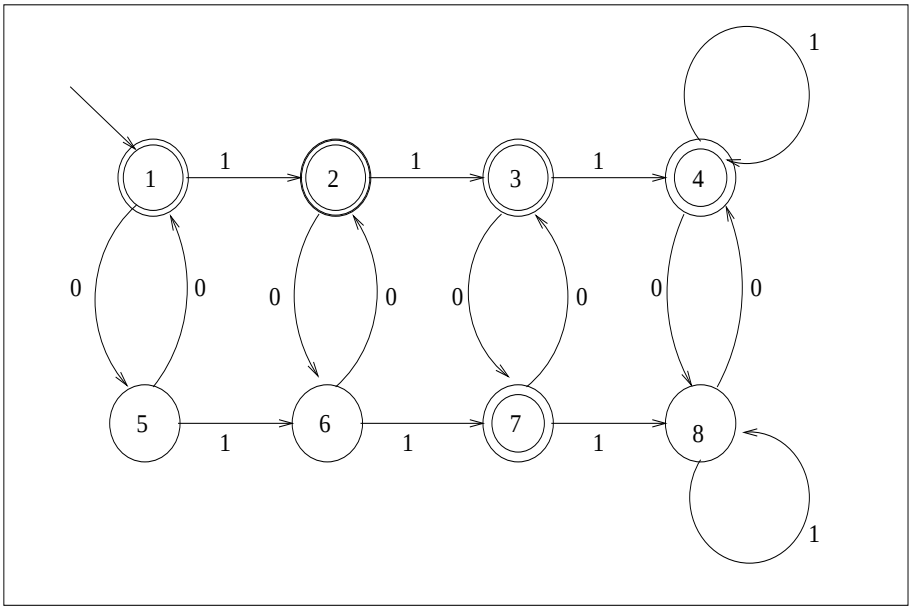


Figure: DFA for exercise 1.4 l

3. Exercise 1.5 a, e.

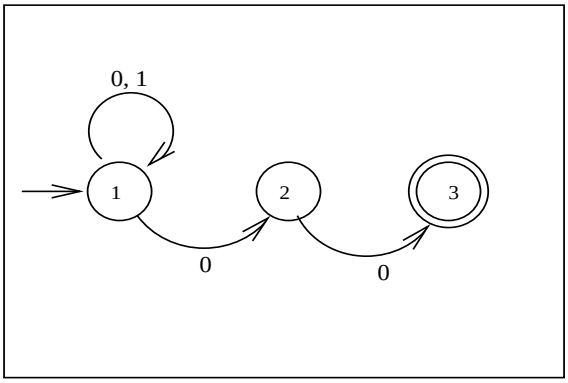


Figure: A 3-state NFA for exercise 1.5 a

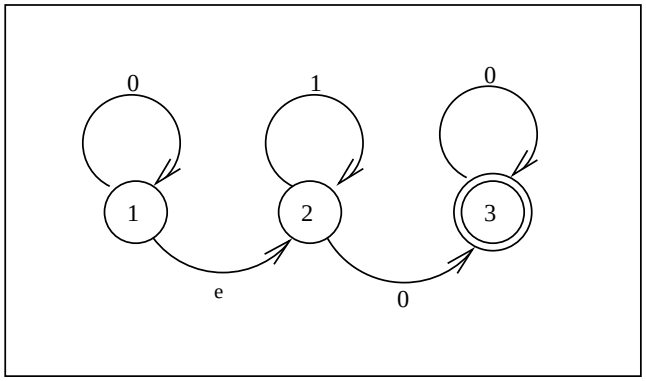


Figure: A 3-state NFA for 0*1*0*0 (e represents ϵ) [exercise 1.5 e]

---

4. Exercise 1.10 b.

Consider the following NFAs. The one on the left recognizes the language $\{0\}$. The one on the right results from swapping the accept and non-accept states of the first NFA. It recognizes the language $\{\varepsilon\}$. Notice that $\{\varepsilon\} \neq \overline{\{0\}}$.

The class of languages recognized by NFAs is the class of regular languages. In class we proved that this class is closed under complement. Observe that this is not a contradiction. The example above shows that the construction we used to prove that the class of regular languages is closed under complement doesn't work for NFAs. But, given an NFA that recognizes a language C, one can construct an NFA that recognizes $\overline{C}$. For example, one can first construct a DFA that is equivalent

to the given NFA, and then swap its accept and non-accept states. The result is an NFA that recognizes $\overline{C}$.

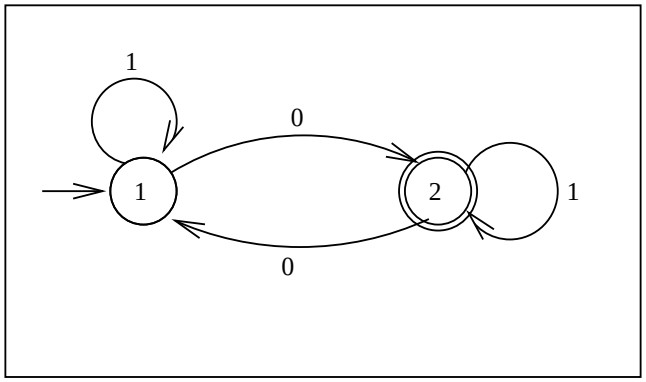
---

5. Exercise 1.11.

Consider the following language:

$$L = \{\omega \in \{0,1\}^* | \ \omega \text{ contains odd number of 0s}\}$$

The following DFA, N_1 recognizes the above language:



The DFA N_1 which accepts strings with odd number of 0s

Now we construct the following NFA, N, by using the procedure defined in the problem.

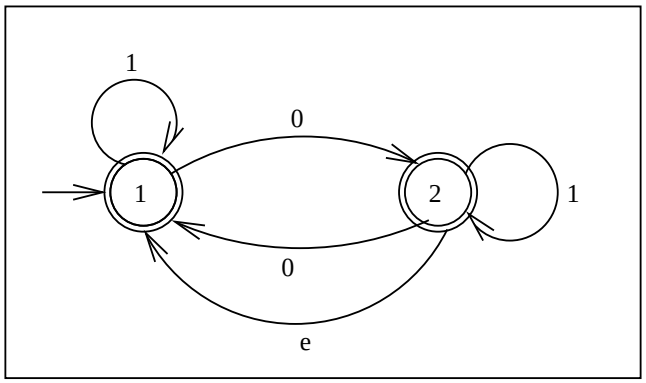


Figure: The DFA N is constructed by making q_1 as a final state and adding an ϵ transition from q_2 to q_1. (e represents ϵ).

We find that $1 \notin L^*$ but N accepts 1. So this construction does *not* show that the class of regular languages is closed under the star operation.

---

6. Exercise 1.12 (a).

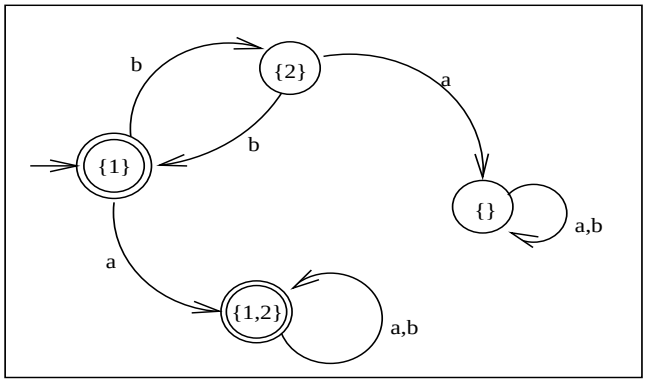


Figure: NFA corresponding to the given DFA

---

7. Exercise 1.13 b, e.

b. $(0 \cup 1)^*1(0 \cup 1)^*1(0 \cup 1)^*1(0 \cup 1)^*$

e. $0((0 \cup 1)(0 \cup 1))^* \cup 1(0 \cup 1)((0 \cup 1)(0 \cup 1))^*$

---

8. Problem 1.31.

The problem is asking us to prove that a language is regular if and only if there exists an all-paths-NFA that recognizes the language. We know that a language is regular if and only if there exists a DFA that recognizes it. So to solve the problem we will show two things:

1) Given an all-paths-NFA, we can construct a DFA that recognizes the same language as the all-paths-NFA, and

2) Given a DFA, we can construct an all-paths-NFA that recognizes the same language as the DFA

For the first part, we perform the conversion from an all-paths-NFA to a DFA in the exact same way as we converted a regular NFA to a DFA (pg. 56-58 of Sipser) but with one small difference. In the regular NFA conversion, the accept states of the DFA are those that contain at least one of the NFA's accept states. But when converting from an all-paths-NFA, the accept states of the DFA are those that contain *only* the all-paths-NFA's accept states. Thus, given an all-paths-NFA, we can construct a DFA that recognizes the same language.

The second part is easier. If we have a DFA, then an all-paths-NFA that recognizes the same language is exactly the same DFA! Why? Because every DFA has exactly one path of computation for each string. If the string is in the language, then the path leads to an accept state, and if it's not in the language, then the path leads to a non-accept state. Thus if a string is in the language, then all paths (the one path) lead to an accept state. So it's an all-paths-NFA for the language.

By proving 1) and 2), we showed that the set of languages recognized by all-paths-NFAs is the same as the set of languages recognized by DFAs. Therefore the set of languages recognized by all-paths-NFAs is the set of regular languages.

9. Problem 1.32.

a. **Given:** A is regular, i.e., there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes A.

Want: To show that

$$\text{NOPREFIX}(A) = \{ w \in A \mid \text{no proper prefix of } w \text{ is a member of } A \}$$

is regular.

Comment: *When we proved closure properties in lecture, to show that the class of regular languages is closed under some unary operation, we used a DFA recognizing a regular language A, to construct an NFA recognizing the language that results from applying the operation to A. Here we use a slighty different approach. We do not construct an NFA that recognizes* NOPREFIX(A). *Instead, we consider a related language L for which we can* **easily** *build an NFA N that recognizes L, and then we use the fact that L is regular to prove that* NOPREFIX(A) *is regular.*

Construction and correctness: Consider the following language:

$$\begin{aligned} L &= \{ w \in \Sigma^* \mid \text{some proper prefix of } w \text{ is a member of } A \} \\ &= \{ w \in \Sigma^* \mid \text{some string } x \in A \text{ is a proper prefix of } w \} \end{aligned}$$

We construct an NFA $N = (Q', \Sigma', \delta', q_0', F')$ that recognizes L as follows:

- $Q' = Q \cup \{q_f\}$, where $q_f \notin Q$ is a new state
- $\Sigma' = \Sigma$
- δ' is defined as follows:

$$\begin{array}{lcll} \delta'(q,\sigma) & = & \{\delta(q,\sigma)\} & \text{if } q \in Q \setminus F,\ \sigma \in \Sigma \\ \delta'(q,\sigma) & = & \{\delta(q,\sigma), q_f\} & \text{if } q \in F,\ \sigma \in \Sigma \\ \delta'(q,\sigma) & = & \{q_f\} & \text{if } q = q_f,\ \sigma \in \Sigma \\ \delta'(q,\sigma) & = & \emptyset & \text{otherwise} \end{array}$$

- $q_0' = q_0$
- $F' = \{q_f\}$

Claim: N recognizes L.

Proof: We must show that if $w \in L$ then N accepts w, and if $w \notin L$ then N rejects w.

If $w \in L$, then there is some string $x \in A$ that is a proper prefix of w, i.e., $w = xy$, where y is not empty. Consider the computation of N on input $w = xy$. When N reads the last symbol in x, it moves to a state corresponding to an accept state in M (by the definition of δ, and because $x \in A$ and M recognizes A). When N reads the first symbol in y, it moves to state q_f (by the definition of δ). It then stays in this state until the input is exhausted. Since q_f is an accept state, N accepts the string w.

If $w \notin L$, then no proper prefix of w is in A. Consider the computation of N on input w. If $w = \varepsilon$, then N rejects. If $w \neq \varepsilon$, then N does not enter a state corresponding to an accept

state in M before reading the last symbol in w (by the definition of δ and the fact that no proper prefix of w is in A). Therefore, when N reads the last symbol in w, it will not move to q_f (the only way to move to q_f is by going through a state corresponding to an accept state in M). Hence N rejects the string w. ▌

Since there exists an NFA that recognizes L, L is regular.

We observe that NOPREFIX$(A) = A \cap \overline{L}$. Since the class of regular languages is closed under complement and intersection, NOPREFIX(A) is regular.

ALTERNATIVE, SIMPLER, SOLUTION:

Comment: *Here we will use a DFA M recognizing the regular language A, to construct an NFA N recognizing the language* NOPREFIX(A). *The key observation is that the strings in* NOPREFIX(A) *are those for which the computation path in M contains only one accept state. We build N by removing from M all transitions out of accept states in M.*

Given: A is regular, i.e., there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes A.

Want: To show that

$$\text{NOPREFIX}(A) = \{\, w \in A \mid \text{no proper prefix of } w \text{ is a member of } A \,\}$$

is regular, i.e., there exists an NFA N that recognizes NOPREFIX(A).

Construction: Let $N = (Q', \Sigma', \delta', q_0', F')$, where

- $Q' = Q$
- $\Sigma' = \Sigma$
- δ' is defined as follows:

$$\begin{array}{lcll} \delta'(q,\sigma) & = & \{\delta(q,\sigma)\} & \text{if } q \notin F,\ \sigma \in \Sigma \\ \delta'(q,\sigma) & = & \emptyset & \text{otherwise} \end{array}$$

- $q_0' = q_0$
- $F' = F$

Correctness: We must show that if $w \in$ NOPREFIX(A) then N accepts w, and if $w \notin$ NOPREFIX(A) then N rejects w.

Coming soon.

Comment: *In case you're wondering why we first presented the more involved solution, it's simply because it was the first idea we had. After obtaining a deeper understanding of the operation* NOPREFIX*, the second solution came to mind. This is another example of how it can it pay off to think about the operation or language carefully and understand it before trying to solve the problem.*

b. ***Comment:*** *Here we will use a DFA M recognizing the regular language A, to construct a DFA M' recognizing the language* NOEXTEND(A). *The key observation is that the strings in* NOEXTEND(A) *are those for which the computation path in M reaches an accept state and this path cannot be extended by following more transitions to arrive again at an accept state.*

We build M' by removing from the set of accept states in M those that can reach an accept state by following one or more transitions.

Given: A is regular, i.e., there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes A.

Want: To show that

$$\text{NOEXTEND}(A) \quad = \quad \{\, w \in A \mid w \text{ is not a proper prefix of any string in } A \,\}$$

is regular, i.e., there exists an NFA that recognizes NOEXTEND(A).

Construction: We construct a DFA $M' = (Q', \Sigma', \delta', q_0', F')$ that recognizes NOEXTEND(A) as follows:

- $Q' = Q$
- $\Sigma' = \Sigma$
- $\delta' = \delta$
- $q_0' = q_0$
- $F' = F \setminus \{\, q \in F \mid \text{there is a path of length at least one from } q \text{ to some } q_f \in F \,\}$

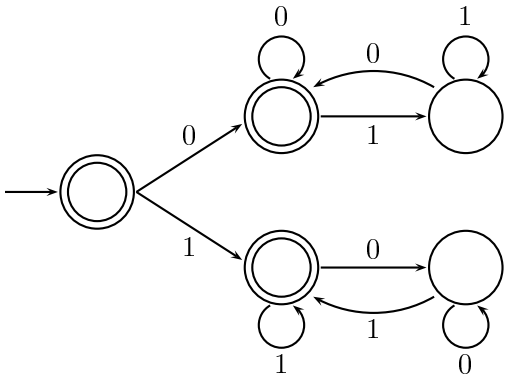
Correctness: We must show that if $w \in$ NOEXTEND(A) then M' accepts w, and if $w \notin$ NOEXTEND(A) then M' rejects w.

Coming soon.

---

10. Problem 1.41.

The following is a DFA that recognizes language D.



---

11. Problem 1.42.

Given: A is regular, i.e., there is a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes A.

Want: To show that

$$A_{\frac{1}{2}-} = \{\, x \mid \text{for some } y, |x| = |y| \text{ and } xy \in A \,\}$$

is regular.

We will do this by constructing an NFA N that accepts $A_{\frac{1}{2}-}$.

Construction: Let $N = (Q', \Sigma', \delta', q_0', F')$, where

- $Q' = (Q \times Q) \cup \{q_s\}$, where $q_s \notin Q$ is a new state
- $\Sigma' = \Sigma$
- δ' is defined as follows:

$$\begin{aligned}
\delta'((r,t),\sigma) &= \{\, (\delta(r,\sigma), q) \mid q \in Q \text{ and there exists } \alpha \in \Sigma \text{ such that } \delta(q,\alpha) = t \,\} \\
&\quad \text{if } (r,t) \in Q \times Q,\ \sigma \in \Sigma \\
\delta'(q_s,\sigma) &= \{\, (q_0, q) \mid q \in F \,\} \quad \text{if } \sigma = \varepsilon \\
\delta'(q,\sigma) &= \emptyset \quad \text{otherwise}
\end{aligned}$$

- $q_0' = q_s$
- $F' = \{\, (q,q) \mid q \in Q \,\}$

We claim that N recognizes $A_{\frac{1}{2}-}$. This must be proved.

Correctness of construction: We must show that if $x \in A_{\frac{1}{2}-}$ then N accepts x, and if $x \notin A_{\frac{1}{2}-}$ then N rejects x.

If $x = x_1 \cdots x_n \in A_{\frac{1}{2}-}$, then there exists some string $y = y_1 \cdots y_n$ such that $xy \in A$. Since M recognizes A, M accepts xy. Consider the path followed in M on this input. It has the form

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \cdots\ q_{n-1} \xrightarrow{x_n} q_n \xrightarrow{y_1} q_{n+1} \xrightarrow{y_2} \cdots\ q_{2n-1} \xrightarrow{y_n} q_{2n}\ ,$$

where $q_0, \ldots, q_{2n}$ are some states in Q, namely the ones followed on this path, and q_{2n} is an accept state. Now consider the path

$$q_s \xrightarrow{\varepsilon} (q_0, q_{2n}) \xrightarrow{x_1} (q_1, q_{2n-1}) \xrightarrow{x_2} \cdots\ (q_{n-1}, q_{n+1}) \xrightarrow{x_n} (q_n, q_n)\ .$$

This is a valid path on input x in the NFA N, and it ends in an accept state of N. Therefore, N accepts x.

We must show that if $x \notin A_{\frac{1}{2}-}$, then there is *no* accepting path in N on input x. We provide a proof by contradiction. Assume that there is an accepting path in N on input x. This path has the form

$$q_s \xrightarrow{\varepsilon} (q_0, q_{2n}) \xrightarrow{x_1} (q_1, q_{2n-1}) \xrightarrow{x_2} \cdots\ (q_{n-1}, q_{n+1}) \xrightarrow{x_n} (q_n, q_n)\ ,$$

where $q_0, \ldots, q_{2n}$ are some states in Q and q_{2n} is an accept state of DFA M. By the definition of the transition function δ', for $i = 1, \ldots, n$, $\delta(q_{i-1}, x_i) = q_i$ and there exists $\alpha_i \in \Sigma$ such that $\delta(q_{n+i-1}, \alpha_i) = q_{n+i}$. Now consider the path

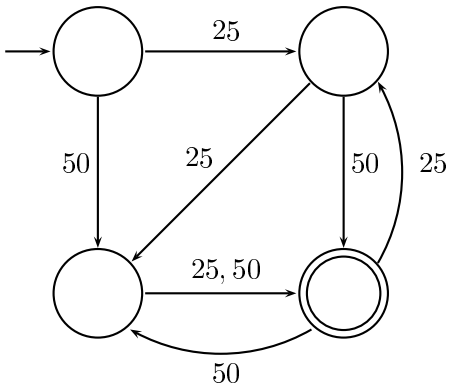
$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \cdots\ q_{n-1} \xrightarrow{x_n} q_n \xrightarrow{\alpha_1} q_{n+1} \xrightarrow{\alpha_2} \cdots\ q_{2n-1} \xrightarrow{\alpha_n} q_{2n}\ .$$

This is a valid path on input $x\alpha_1 \cdots \alpha_n$ in the DFA M, and it ends in an accept state of M. Therefore, M accepts $x\alpha_1 \cdots \alpha_n$. But, $|\alpha_1 \cdots \alpha_n| = |x|$ and since $x \notin A_{\frac{1}{2}-}$ we know that for all y

such that $|y| = |x|$, $xy \notin A$, so $x\alpha_1 \cdots \alpha_n \notin A$. Thus M accepts a string that is not in A. This contradicts the assumption that M recognizes A.

---

12. Consider the following DFA, where the alphabet is $\{25, 50\}$:



Assume this DFA is used to implement a vending machine that accepts only quarters and half-dollars, sells cans of Coke for \$0.75 each, and does not return change. Assume there is a sign on the machine that says "This machine does not return change."

a) Assume the vending machine is activated when a coin is inserted, and it waits for additional coins to be inserted until the customer pushes a button. The DFA processes its input as the coins are inserted, and the machine keeps track of the number of times the DFA enters an accept state – say n. When the customer pushes the button to obtain the Coke, n cans are dispensed by the machine. Give an example of an input for which the customer will be surprised to lose money.

Inserting 50,50,50, you have enough for 2 cans of soda, but the machine will only give you 1.

b) What language is recognized by this DFA?

Notice that the start state and the accept state of the DFA are virtually identical (except that one is an accept state and the other is not). The reason that they are identical is they both go to the lower left state if a 50 cent coin is inserted and they both go to the upper right state if a 25 cent coin is inserted. So every time that the customer puts in enough money to get one soda (be it 75 cents or 1 dollar), the machine gives out a soda and then "resets". So the language recognized by the DFA is where there is no money left over in the machine at the end of your transaction when you buy the sodas as specified above (i.e. you put in either 75 cents or 1 dollar, and the machine resets to 0).

c) Give a regular expression describing the language in your answer to **b)**.

Represent 25 cent coins by q and 50 cent coins by h. The regular expression is then $((qqq) \cup (qh) \cup (hq) \cup (qqh) \cup (hh))^+$.

d) Now assume the vending machine waits for additional coins to be inserted only until the DFA enters an accept state. Then it immediately dispenses a can, and it does not allow more coins to be inserted until the can is dispensed. Are there inputs for which the customer will be

surprised to lose money? If so, give an example.

There is no input for which the customer will feel surprised that he lost money. In part (a), the customer can put in 3 coins 50 cents each and he may think that he has enough for 2 cans of soda since he has no idea what the DFA inside the vending machine looks like (it might have been a smarter DFA that would actually give him 2 cans of soda) and thus may feel surprised when he only receives one. But if he gets a can of soda as soon as he puts in two 50 cent coins, and the vending machine states that it does not give change, he should not be surprised that the machine kept the 25 cents change for itself and thus he would need to put in at least 75 cents more to get another can of soda. Since the customer gets a can of soda as soon as his total reaches 75 cents, he should never feel surprised at what the machine does.

---

13. Exercise 1.15 b, d, g.

b. In the language: ab, abab. Not in the language: a, b.

d. In the language: aaa, aaaaaa. Not in the language: a, aa.

g. In the language: b, ab. Not in the language: a, aab.

---

14. Let Σ be an alphabet. Given a string $w = w_1 w_2 \cdots w_n$, where $w_i \in \Sigma$ for $i = 1, 2, \ldots, n$, let $even(w) = w_2 w_4 \cdots w_{n-(n \bmod 2)}$ denote the string that results from keeping only the symbols at even positions in w, and let $odd(w) = w_1 w_3 \cdots w_{n-1+(n \bmod 2)}$ denote the string that results from keeping only the symbols at odd positions in w. For example, for $\Sigma = \{0, 1\}$, we have

w	$even(w)$	$odd(w)$
0101010101	11111	00000
11111	11	111
001110110	0101	01110
0	ε	0
ε	ε	ε

The operations *even* and *odd* can be extended to languages as follows:

$$even(L) = \{\, even(w) \mid w \in L \,\} \qquad odd(L) = \{\, odd(w) \mid w \in L \,\}$$

For example, for $\Sigma = \{a, \ldots, z\}$, we have

L	$even(L)$	$odd(L)$
$\{shdjaig, ndkgyzew, sjdhl\}$	$\{hji, dgzw, jh\}$	$\{sdag, nkye, sdl\}$
$\{a, bcd\}$	$\{\varepsilon, c\}$	$\{a, bd\}$
$\{\, w \in \Sigma^* \mid \lvert w \rvert$ is even $\}$	Σ^*	Σ^*
Σ^*	Σ^*	Σ^*
$\emptyset$	$\emptyset$	$\emptyset$

Prove that if L is a regular language, then $even(L)$ and $odd(L)$ are both regular.

---

15. Problem 1.17 c.

Proof by contradiction. Assume A_3 is regular, and therefore the pumping lemma holds. Let p be the pumping length given by the pumping lemma. Let $s = a^{2^p}$. Because s is in A_3 and has length greater than p, the pumping lemma states that s can be split into three strings, xyz that obey the lemma's conditions. We will show that it is not possible to break s down this way, which contradicts the claim that A_3 is regular.

Let's examine the possibilities of what y may look like. Because s is composed only of a's, y must also contain only a's. Also, the length of y must be at least 1, and y cannot be longer than p (since $|xy| \leq p$). Therefore, the length of y must fall in the range of 1 to p.

Now that we know the properties of all y's that could obey the pumping lemma, we'll show that none of these y's can obey condition three of the pumping lemma. If we choose $i = 2$, the string $s' = xy^2z = xyyz$ must be in the language A_3 if the pumping lemma holds. We know that s' is the same as s, but with the length of y more a's in it. This brings the total number of a's in s' to $2^p + |y|$. The next larger string in A_3 has a number of a's equal to $2^{p+1} = 2 * 2^p$, which is larger than s' could possibly be. This proves that s' is not in A_3. Because s' violates the pumping lemma for any choice of xyz, the claim that A_3 is regular is a contradiction.

---

16. Problem 1.23 c, d.

c. Proof by contradiction. Assume that the language $L = \{0^m1^n | m \neq n\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $0^p1^{p!+p}$. No matter how we break this string up into xyz, y will contain only 0's since $|xy| \leq p$. Breaking s into xyz will look as follows: $x = 0^j$, $y = 0^k$, $z = 0^l1^{p!+p}$ where $j + k + l = p$ and $k \geq 1$. Now we will pump y so that the number of 0's will equal the number of 1's and we'll get a contradiction. Let i be the number of times we pump y. Thus the number of 0's will be $j + ik + l$. We want that to equal to the number of 1's, which is $p! + p$. Now we will solve for i.
$j + ik + l = p! + p$
$j + k(i-1) + k + l = p! + p$
$j + k + l + k(i-1) = p! + p$
$p + k(i-1) = p! + p$
$i - 1 = \frac{p!}{k}$
$i = \frac{p!}{k} + 1$
Since $1 \leq k \leq p$, $\frac{p!}{k}$ will always be a positive integer, and thus i will be a positive integer. Therefore the string s can be pumped so the resulting string is not in the language. Contradiction. Thus L is not regular.

d. Proof by contradiction. Assume that the language $L = \{w | w$ is not a palindrome $\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $0^p10^{p!+p}$. No matter how we break this string up into xyz, y will contain only 0's since $|xy| \leq p$. Breaking s into xyz will look as follows: $x = 0^j$, $y = 0^k$, $z = 0^l10^{p!+p}$ where $j + k + l = p$ and $k \geq 1$. Now we will pump y so that the number of 0's in the beginning will equal the number of 0's at

the end, and so the resulting string will be a palindrome and thus not in the language. Let i be the number of times we pump y. Thus the number of 0's in the beginning will be $j+ik+l$. We want that to equal to the number of 0's at the end which is $p!+p$. Now we solve for i.
$j+ik+l=p!+p$
$j+k(i-1)+k+l=p!+p$
$j+k+l+k(i-1)=p!+p$
$p+k(i-1)=p!+p$
$i-1=\frac{p!}{k}$
$i=\frac{p!}{k}+1$
Since $1 \leq k \leq p$, $\frac{p!}{k}$ will always be a positive integer, and thus i will be a positive integer. Therefore the string s can be pumped so the resulting string is not in the language. Contradiction. Thus L is not regular.

An easier way to prove that L is not regular in this case is to prove that $\bar{L}$ is not regular.
$\bar{L}=\{w|w$ is a palindrome $\}$.
Proof by contradiction. Assume that the language is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string 0^p10^p. No matter how we break this string up into xyz, y will contain only 0's since $|xy| \leq p$. Breaking s into xyz will look as follows: $x=0^j$, $y=0^k$, $z=0^l10^p$ where $j+k+l=p$ and $k \geq 1$. Now we just pump y once. So the resulting string is $xyyz$ which is $0^j0^{2k}0^l10^p$. The number of 0's in the beginning is $j+2k+l=p+k>p$. The number of 0's at the end is still p. Thus the resulting string is not a palindrome and thus not in the language. Contradiction. Therefore the language $\bar{L}$ is not regular. Since $\bar{L}$ is not regular, neither is L.

---

17. Prove that the converse of the result of problem **14.** above is not necessarily true, i.e., show that there exists a language L such that both $even(L)$ and $odd(L)$ are regular, but L is not regular. [Hint: One of the languages from Problem 1.23 in the textbook will do.] Notice that once you choose a language L, you must prove that $even(L)$ and $odd(L)$ are regular (for example, by giving a regular expression or NFA for each of them, or by using closure properties of the class of regular languages) *and* prove that L is not regular (by using the pumping lemma or closure properties of the class of regular languages).

---

18. Exercise 2.1.

a. $E \Rightarrow T \Rightarrow F \Rightarrow a$

b. $E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+F \Rightarrow a+a$.

c. $E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow T+T+T \Rightarrow F+T+T \Rightarrow a+T+T \Rightarrow a+F+T \Rightarrow a+a+T \Rightarrow a+a+F \Rightarrow a+a+a$

d. $E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow ((E)) \Rightarrow ((T)) \Rightarrow ((F)) \Rightarrow ((a))$.

---

19. Exercise 2.3 a, b, ... , m.

a. The variables are $\{R, S, T, X\}$. The terminals are $\{a, b\}$. The start variable is R.

b. $\{ab, ba, aab\}$

c. $\{\epsilon, aa, bb\}$

d. False

e. True

f. False

g. True

h. True

i. False

j. True

k. True

l. False

m. All strings over $\{a, b\}$ that are not palindromes.

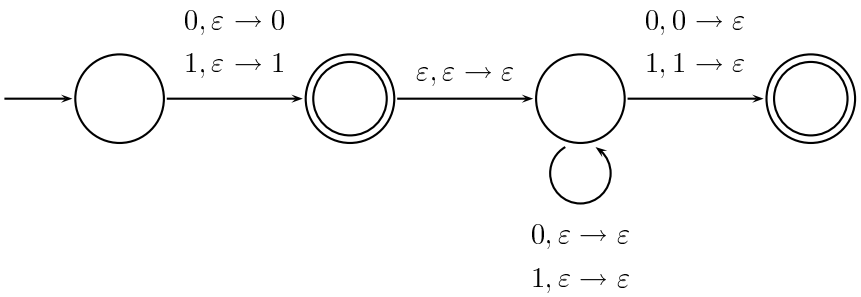
---

20. Exercise 2.4 b, f.

---

21. Exercise 2.5 b, f.

Since this language is regular, providing a PDA that recognizes the language is easy. We can design a DFA or NFA that recognizes it. A PDA is an NFA with a stack, so it is easy to convert a DFA or NFA to a PDA: we just program the PDA to ignore its stack. (It is not true that an NFA is PDA, but it is true that an NFA can be easily converted to a PDA that recognizes the same language as the original NFA.) However, this solution isn't much fun, so instead we will provide a PDA that does exploit its stack, and ends up being simpler than the PDA one obtains via a DFA or NFA. The idea is to push the first symbol of the input w on the stack, and then accept if the last symbol of w matches it. The state diagram of the resulting PDA is below.



---

22. Exercise 2.6 b, c.

---

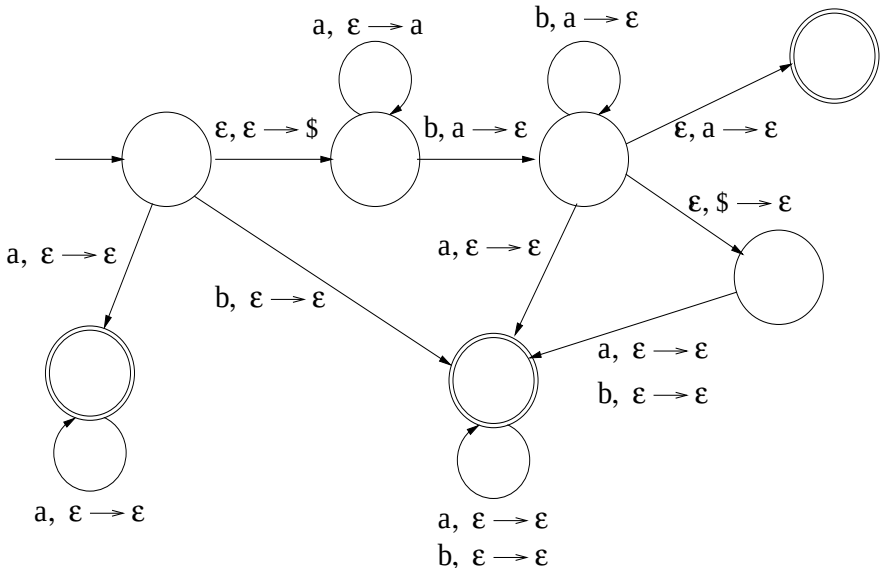
23. Exercise 2.7 b, c. Give state diagrams also.

b. Let us assume $\Sigma = \{a, b\}$ (the solution given below can be easily extended for more general

Σ). The strings in the complement of $\{a^n b^n | n \geq 0\}$ i.e. $\Sigma^* \setminus \{a^n b^n | n \geq 0\}$ can be put into four catagories:

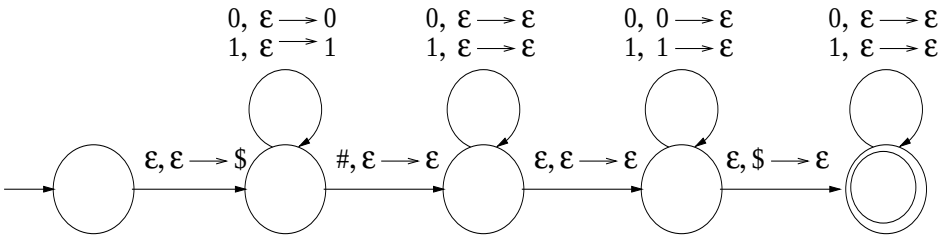
1. Any string beginning with a b is in the language
2. Any string consisting only of a's is in the language
3. Any string of the form $a^n b^m$ $(m < n)$ is in the language
4. Any string of the form $a^n b^m a w$ $(m < n)$ where w is any string in Σ^* is in the language; and
5. Strings of the form $a^n b^n w$ where w is any non-empty string in Σ^* are also in the language.

Our PDA should then be designed as follows. If it sees a string that starts with a b, it directly enters an accepting state and keeps looping on it on any further input, without making any changes to the stack (this corresponds to Case 1 above). If, on the other hand, the first symbol that it sees is an a, it can follow four possible paths. The first path, the simplest of all, takes it to an accepting state on which it keeps looping on any further a's that it sees (again, without modifying the stack in any way). This corresponds to Case 2 above. In the second, third and fourth paths, it counts the number of a's that it sees by pushing a symbol onto the stack for every input a. Then when it sees a b, the PDA starts popping off the symbols that it had pushed, one symbol for every b that it sees. If the string ends even before it has popped off all symbols (Case 3), it goes into an accepting state. Or else, if it manages to see an a before all the stack symbols have been popped off (Case 4), it goes into another accepting state and remains there for any further input that it sees. If, however, it does manage to pop off all the symbols while reading the b's, it goes into a new state in which it waits for at least one other symbol in the input (any symbol, an a or a b, would do). The moment it sees another symbol coming in (Case 5), it again goes into an accepting state and keeps looping on it upon any further input. The following figure shows the state diagram for the PDA:



c. The solution to this problem draws ideas from Example 2.11 given in the book. Our PDA first places a bottom marker, \$, onto the stack and, subsequently, upon reading any 1 from

the input it pushes a 1 onto the stack and upon reading a 0, it pushes a 0. It keeps doing this until it sees the center marker, #. This helps it keep track of the string, w, preceding the marker (if $w = \epsilon$, nothing gets pushed on the stack after the \$, which is fine). Once it has seen the marker it goes into a new state in which it reads 0 or more symbols (from $\{0, 1\}$) without making any changes to the stack. At some later point, it enters a new state in which it pops off the symbols off the stack one by one, such that at every step, the symbol removed from the stack is the same as the input symbol being currently read. Once it has popped off all symbols it placed on the stack (indicating that the string $w^{\mathcal{R}}$ has been read completely), it goes into an accepting state in which it keeps looping on any further input. The following figure shows the state diagram for the PDA.



---

24. Exercise 2.8

The first derivation of the sentence is as follows:

$$
\begin{aligned}
\langle SENTENCE \rangle &\Rightarrow \langle NOUN-PHRASE \rangle \langle VERB-PHRASE \rangle \\
&\Rightarrow \langle CMPLX-NOUN \rangle \langle VERB-PHRASE \rangle \\
&\Rightarrow \langle ARTICLE \rangle \langle NOUN \rangle \langle VERB-PHRASE \rangle \\
&\Rightarrow \texttt{the} \langle NOUN \rangle \langle VERB-PHRASE \rangle \\
&\Rightarrow \texttt{the girl} \langle VERB-PHRASE \rangle \\
&\Rightarrow \texttt{the girl} \langle CMPLX-VERB \rangle \\
&\Rightarrow \texttt{the girl} \langle VERB \rangle \langle NOUN-PHRASE \rangle \\
&\Rightarrow \texttt{the girl touches} \langle NOUN-PHRASE \rangle \\
&\Rightarrow \texttt{the girl touches} \langle CMPLX-NOUN \rangle \langle PREP-PHRASE \rangle \\
&\Rightarrow \texttt{the girl touches} \langle ARTICLE \rangle \langle NOUN \rangle \langle PREP-PHRASE \rangle \\
&\Rightarrow \texttt{the girl touches the} \langle NOUN \rangle \langle PREP-PHRASE \rangle \\
&\Rightarrow \texttt{the girl touches the boy} \langle PREP-PHRASE \rangle \\
&\Rightarrow \texttt{the girl touches the boy} \langle PREP \rangle \langle CMPLX-NOUN \rangle \\
&\Rightarrow \texttt{the girl touches the boy with} \langle CMPLX-NOUN \rangle \\
&\Rightarrow \texttt{the girl touches the boy with} \langle ARTICLE \rangle \langle NOUN \rangle \\
&\Rightarrow \texttt{the girl touches the boy with the} \langle NOUN \rangle
\end{aligned}
$$

$$\Rightarrow \texttt{ the girl touches the boy with the flower}$$

This derivation corresponds to the interpretation that the flower is something that the "boy" has (for example, the boy could be holding the flower in his hand). How the girl touches him (with her hand, with her elbow, with another flower etc) is unspecified by this interpretation.

The second derivation works as follows:

$$\begin{aligned}
\langle SENTENCE\rangle &\Rightarrow \langle NOUN-PHRASE\rangle\langle VERB-PHRASE\rangle \\
&\Rightarrow \langle CMPLX-NOUN\rangle\langle VERB-PHRASE\rangle \\
&\Rightarrow \langle ARTICLE\rangle\langle NOUN\rangle\langle VERB-PHRASE\rangle \\
&\Rightarrow \texttt{the}\langle NOUN\rangle\langle VERB-PHRASE\rangle \\
&\Rightarrow \texttt{the girl}\langle VERB-PHRASE\rangle \\
&\Rightarrow \texttt{the girl}\langle CMPLX-VERB\rangle\langle PREP-PHRASE\rangle \\
&\Rightarrow \texttt{the girl}\langle VERB\rangle\langle NOUN-PHRASE\rangle\langle PREP-PHRASE\rangle \\
&\Rightarrow \texttt{the girl touches}\langle NOUN-PHRASE\rangle\langle PREP-PHRASE\rangle \\
&\Rightarrow \texttt{the girl touches}\langle CMPLX-NOUN\rangle\langle PREP-PHRASE\rangle \\
&\Rightarrow \texttt{the girl touches}\langle ARTICLE\rangle\langle NOUN\rangle\langle PREP-PHRASE\rangle \\
&\Rightarrow \texttt{the girl touches the}\langle NOUN\rangle\langle PREP-PHRASE\rangle \\
&\Rightarrow \texttt{the girl touches the boy}\langle PREP-PHRASE\rangle \\
&\Rightarrow \texttt{the girl touches the boy}\langle PREP\rangle\langle CMPLX-NOUN\rangle \\
&\Rightarrow \texttt{the girl touches the boy with}\langle CMPLX-NOUN\rangle \\
&\Rightarrow \texttt{the girl touches the boy with}\langle ARTICLE\rangle\langle NOUN\rangle \\
&\Rightarrow \texttt{the girl touches the boy with the}\langle NOUN\rangle \\
&\Rightarrow \texttt{the girl touches the boy with the flower}
\end{aligned}$$

This derivation corresponds to the interpretation that the flower is something that the "girl" is using to carry out her action (i.e. to touch the boy).

---

25. Problem 2.17 a.

Given: C is context-free, i.e., there is a PDA $M_C = (Q_C, \Sigma, \Gamma, \delta_C, q_C, F_C)$ that recognizes C. Also R is regular, i.e., there is a DFA $M_R = (Q_R, \Sigma, \delta_R, q_R, F_R)$ that recognizes R.

Want: To construct a PDA M that recognizes $C \cap R$, showing that $C \cap R$ is a context-free language.

Construction: The idea of the construction is that M can run M_C and M_R "in parallel", using its stack to simulate the stack of M_C. Parallelism is implemented by making a state of M a pair consisting of a state of M_C and a state of M_R.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- $Q = Q_C \times Q_R$ (that is, a state of M is a pair (c, r) where $c \in Q_C$ is a state of M_C and $r \in M_R$ is a state of M_R)
- $q_0 = (q_C, q_R)$ (that is, the start state of M is the pair consisting of the start state of M_C and the start state of M_R)
- $F = F_C \times F_R$ (that is, a final state of M is a pair (c, r) where $c \in F_C$ is a final state of M_C and $r \in F_R$ is a final state of M_R).
- The transition function δ takes input a state $(c, r) \in Q$, an input symbol $\sigma \in \Sigma_\varepsilon$ and a top-of-stack symbol $\gamma \in \Gamma_\varepsilon$, and returns the following:

$$\delta((c,r),\sigma,\gamma) = \begin{cases} \{((c', \delta_R(r,\sigma)), \gamma')\}(c', \gamma') \in \delta_C(c,\sigma,\gamma) & \text{if } \sigma \neq \varepsilon \\ \{((c', r), \gamma')\}(c', \gamma') \in \delta_C(c,\sigma,\gamma) & \text{if } \sigma = \varepsilon \end{cases}$$

Notice that the stack alphabet of M is the same as the stack alphabet of M_C, and the input alphabets of M, M_C, M_R are all the same.

Observe that if PDA M_C was in state c, scanning input symbol σ, and γ was on top of its stack, then its transition function δ_C tells us that it could go to state c' and push γ' on top of the stack for any (c', γ') in the set $\delta_C(c, \sigma, \gamma)$. Our machine M starts from a state of the form (c, r) where r is state of M_R. It wants to update c and its stack according to δ_C, while simultaneously updating r as per the transition function of M_R, meaning to the value $\delta_R(r, \sigma)$. That's what the above does, except that it is careful to make a special case when M_C has an ε-transition. In that case, c and the stack should be updated, but r stays the same.

---

26. Problem 2.25

Comment: *The solution for this problem has been written in discrete steps that could be followed in proceeding towards the solution. Our recommendation is that a reader who has only partially succeeded in solving this problem should view each step as a hint and try to get the complete solution hint by hint.*

(1) Your first step should be to figure out the kind of strings that can be derived from the variable Y alone (this is because the strings that are contained in the language are related to whatever can be derived out of Y). If you spend some time thinking, you should be able to see that *every* possible string in the set $\{a, b\}^*$ (including ϵ) can be derived from Y (try writing out the strings that can be obtained by applying the rules for Y and you'll see this).

(2) Using the above observation, work out the kind of strings that can be derived from S. Look at the simpler rules first. Any string starting in a b can be derived from S (using the rule $S \to bY$). Any string ending in an a can be derived from S (using the rule $S \to Ya$). What are the strings in $\{a, b\}^*$ that are not covered by these two kinds of strings ? The strings which both start in an a and end in a b and the string ϵ. We conclude that the language $L(G)$ must *at least* contain any string which is not of the form axb (where $x \in \{a, b\}^*$). Also observe that the string ϵ cannot be derived from S by the application of any of the 3 rules for S.

(3) But hey ! Look at the first rule for S. It says $S \to aSb$ meaning that some strings starting in an a and ending in a b *can* be derived from S. But can all such strings in $\{a, b\}^*$ be derived from it ? That in turn depends on what S can derive. We saw in step 2 above that any string

that isn't ϵ and that doesn't start in an a and end in a b can surely be derived from S. The rule $S \rightarrow aSb$ gives us the impression that a lot many strings beginning with an a and ending in a b can also be derived from S. Which ones can't be derived ? Well, can you derive the string ab from S ? NO. Since we can't derive the empty string from S we can't use the rule $S \rightarrow aSb$ in deriving the string ab. Similarly, can you derive the string $aabb$ from S ? NO. Since we can't derive ab from S, we can't use the rule $S \rightarrow aSb$ in order to derive $aabb$.

(4) Applying this argument recursively, we conclude that any string which is NOT formed by concatenating a sequence of a's with a sequence of b's of *equal length* cannot be derived from S. Mathematically, this means $L(G) = \{a, b\}^* \setminus \{a^n b^n | n \geq 0\}$. Note that this notation captures the fact that ϵ is also not in the language.

(5) What's the complement of $L(G)$? The language $\{a^n b^n | n \geq 0\}$. We've seen this (and similar) languages a lot many times in class. The grammar for it is simple and the rules are as follows:

$$S \quad \rightarrow \quad aSb|\epsilon$$

---

27. Prove that the class of context-free languages is closed under the concatenation operation.

Given: Languages A_1 and A_2 are context-free. So there is a CFG $G_1 = (V_1, \Sigma_1, R_1, S_1)$ that generates A_1 and a CFG $G_2 = (V_2, \Sigma_2, R_2, S_2)$ that generates A_2.

Want: To construct a CFG $G = (V, \Sigma, R, S)$ that generates the language $A_1 \cdot A_2$, showing that $A_1 \cdot A_2$ is a context-free language.

Construction: Grammar $G = (V, \Sigma, R, S)$ is defined as follows:

- $V = V_1 \cup V_2 \cup \{S\}$ where S is a new variable
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $R = R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}$ (the set of rules contains the rules of G_1, the rules of G_2, and one new rule, namely the rule $S \rightarrow S_1 S_2$)

In this construction, we assume that $V_1 \cap V_2 = \emptyset$, meaning that the two given grammars have no variables in common. This can always be made true by renaming variables in one of the grammars if necessary.

Correctness of construction: We claim that $L(G) = L(G_1) \cdot L(G_2)$. (Since $L(G_1) = A_1$ and $L(G_2) = A_2$ this means that $L(G) = A_1 \cdot A_2$, so we are done.) In other words, we claim that $S \Rightarrow^* w$ in G if and only if w has the form $w_1 w_2$ for some w_1, w_2 such that $S_1 \Rightarrow^* w_1$ in G_1 and $S_2 \Rightarrow^* w_2$ in G_2. This is true because the first rule applied in a derivation of any w via G is $S \rightarrow S_1 S_2$, and after that, the only way the derivation can proceed is to expand S_1 according to G_1 and S_2 according to G_2. The fact that $V_1 \cap V_2 = \emptyset$ is important to ensure that a derivation starting from, say, S_1, can only use rules in G_1 and not use rules in G_2, and vice-versa.