Automata, Computability and Complexity with Applications

Exercises in the Book

Solutions

Elaine Rich

Part I: Introduction

1 Why Study Automata Theory?

2 Languages and Strings

1) Consider the language $L = \{1^n 2^n : n > 0\}$. Is the string 122 in L?

No. Every string in L must have the same number of 1's as 2's.

2) Let $L_1 = \{a^n b^n : n > 0\}$. Let $L_2 = \{c^n : n > 0\}$. For each of the following strings, state whether or not it is an element of $L_1 L_2$:

a) ε. No.
b) aabbcc. Yes.
c) abbcc. No.
d) aabbcccc. Yes.

3) Let L_1 = {peach, apple, cherry} and L_2 = {pie, cobbler, ε}. List the elements of $L_1 L_2$ in lexicographic order.

apple, peach, cherry, applepie, peachpie, cherrypie, applecobbler, peachcobbler, cherrycobbler (We list the items shortest first. Within a given length, we list the items alphabetically.)

4) Let $L = \{w \in \{a, b\}^* : |w| \equiv_3 0\}$. List the first six elements in a lexicographic enumeration of L.

ε, aaa, aab, aba, abb, baa

5) Consider the language L of all strings drawn from the alphabet {a, b} with at least two different substrings of length 2.

a) Describe L by writing a sentence of the form $L = \{w \in \Sigma^* : P(w)\}$, where Σ is a set of symbols and P is a first-order logic formula. You may use the function $|s|$ to return the length of s. You may use all the standard relational symbols (e.g., =, ≠, <, etc.), plus the predicate *Substr*(s, t), which is *True* iff s is a substring of t.

$L = \{w \in \{a, b\}^* : \exists x, y\ (x \neq y \wedge |x| = 2 \wedge |y| = 2 \wedge Substr(x, w) \wedge Substr(y, w))\}$.

b) List the first six elements of a lexicographic enumeration of L.

aab, aba, abb, baa, bab, bba

6) For each of the following languages L, give a simple English description. Show two strings that are in L and two that are not (unless there are fewer than two strings in L or two not in L, in which case show as many as possible).

a) $L = \{w \in \{a, b\}^* :$ exactly one prefix of w ends in a$\}$.

L is the set of strings composed of zero or more b's and a single a. a, bba and bbab are in L. bbb and aaa are not.

b) $L = \{w \in \{a, b\}^* :$ all prefixes of w end in a$\}$.

$L = \varnothing$, since ε is a prefix of every string and it doesn't end in a. So all strings are not in L, including a and aa.

c) $L = \{w \in \{a, b\}^* : \exists x \in \{a, b\}^+ (w = axa)\}$.

L is the set of strings over the alphabet {a, b} whose length is at least 3 and that start and end with a. aba, and aaa are in L. ε, a, ab and aa are not.

7) Are the following sets closed under the following operations? If not, what are their respective closures?

a) The language {a, b} under concatenation.

Not closed. $\{w \in \{a, b\}^* : |w| > 0\}$

b) The odd length strings over the alphabet {a, b} under Kleene star.

Not closed because, if two odd length strings are concatenated, the result is of even length. The closure is the set of all nonempty strings drawn from the alphabet {a, b}.

c) $L = \{w \in \{a, b\}^*\}$ under reverse.

Closed. L includes all strings of a's and b's, so, since reverse must also generate strings of a's and b's, any resulting string must have been in the original set.

d) $L = \{w \in \{a, b\}^* : w$ starts with a$\}$ under reverse.

Not closed. L includes strings that end in b. When such strings are reversed, they start with b, so they are not in L. But, when any string in L is reversed, it ends in a. So the closure is $\{w \in \{a, b\}^* : w$ starts with a$\} \cup \{w \in \{a, b\}^* : w$ ends with a$\}$.

e) $L = \{w \in \{a, b\}^* : w$ ends in a$\}$ under concatenation.

Closed.

8) For each of the following statements, state whether it is *True* or *False*. Prove your answer.

a) $\forall L_1, L_2 (L_1 = L_2$ iff $L_1^* = L_2^*)$.

False. Counterexample: $L_1 = \{a\}$. $L_2 = \{a\}^*$. But $L_1^* = L_2^* = \{a\}^* \neq \{a\}$.

b) $(\varnothing \cup \varnothing^*) \cap (\neg\varnothing - (\varnothing\varnothing^*)) = \varnothing$ (where $\neg\varnothing$ is the complement of $\varnothing$).

False. The left hand side equals {ε}, which is not equal to ∅.

c) Every infinite language is the complement of a finite language.

False. Counterexample: Given some nonempty alphabet Σ, the set of all even length strings is an infinite language. Its complement is the set of all odd length strings, which is also infinite.

d) $\forall L ((L^R)^R = L)$.

True.

e) $\forall L_1, L_2 ((L_1 L_2)^* = L_1^* L_2^*)$.

False. Counterexample: $L_1 = \{a\}$. $L_2 = \{b\}$. $(L_1 L_2)^* = (ab)^*$. $L_1^* L_2^* = a^*b^*$.

f) $\forall L_1, L_2\ ((L_1{*}L_2{*}L_1{*}){*} = (L_2 \cup L_1){*})$.

True.

g) $\forall L_1, L_2\ ((L_1 \cup L_2){*} = L_1{*} \cup L_2{*})$.

False. Counterexample: $L_1 = \{\mathtt{a}\}$. $L_2 = \{\mathtt{b}\}$. $(L_1 \cup L_2){*} = (\mathtt{a} \cup \mathtt{b}){*}$. $L_1{*} \cup L_2{*} = \mathtt{a}{*} \cup \mathtt{b}{*}$.

h) $\forall L_1, L_2, L_3\ ((L_1 \cup L_2)\ L_3 = (L_1\ L_3) \cup (L_2\ L_3))$.

True.

i) $\forall L_1, L_2, L_3\ ((L_1\ L_2) \cup L_3 = (L_1 \cup L_3)\ (L_2 \cup L_3))$.

False. Counterexample: $L_1 = \{\mathtt{a}\}$. $L_2 = \{\mathtt{b}\}$. $L_3 = \{\mathtt{c}\}$.

$(L_1\ L_2) \cup L_3 = \{\mathtt{ab}, \mathtt{c}\}$.

$(L_1 \cup L_3)\ (L_2 \cup L_3) = (\mathtt{a} \cup \mathtt{c})(\mathtt{b} \cup \mathtt{c})$

$= \{\mathtt{ab}, \mathtt{ac}, \mathtt{cb}, \mathtt{cc}\}$

j) $\forall L\ ((L^{+}){*} = L{*})$.

True.

k) $\forall L\ (\varnothing L{*} = \{\varepsilon\})$.

False. For any L, and thus for any $L{*}$, $\varnothing L = \varnothing$.

l) $\forall L\ (\varnothing \cup L^{+} = L{*})$.

False. $\varnothing \cup L^{+} = L^{+}$, but it is not true that $L^{+} = L{*}$ unless L includes ε.

m) $\forall L_1, L_2\ ((L_1 \cup L_2){*} = (L_2 \cup L_1){*})$.

True.

3 The Big Picture: A Language Hierarchy

1) Consider the following problem: Given a digital circuit *C*, does *C* output 1 on all inputs? Describe this problem as a language to be decided.

L = {<*C*> : *C* is a digital circuit that outputs 1 on all inputs}. <*C*> is a string encoding of a circuit *C*.

2) Using the technique we used in Example 3.8 to describe addition, describe square root as a language recognition problem.

SQUARE-ROOT = {*w* of the form : <*integer*$_1$>, <*integer*$_2$>, where *integer*$_2$ is the square root of *integer*$_1$}.

3) Consider the problem of encrypting a password, given an encryption key. Formulate this problem as a language recognition problem.

L = {*x*; *y*; *z* : *x* is a string, *y* is the string encoding of an encryption key, and *z* is the string that results from encrypting *x* using *y*}.

4) Consider the optical character recognition (OCR) problem: Given an array of black and white pixels, and a set of characters, determine which character best matches the pixel array. Formulate this problem as a language recognition problem.

L = {<*A*, *C-list*, *c*> : *A* is an array of pixels, *C-list* is a list of characters, and *c* is an element of *C-list* with the property that *A* is a closer match to *c* than it is to any other element of *C-list*}.

5) Consider the language $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$, discussed in Section 3.3.3. We might consider the following design for a PDA to accept $A^nB^nC^n$: As each `a` is read, push two `a`'s onto the stack. Then pop one `a` for each `b` and one `a` for each `c`. If the input and the stack come out even, accept. Otherwise reject. Why doesn't this work?

This PDA will accept all strings in $A^nB^nC^n$. But it will accept others as well. For example, `aabccc`.

6) Define a PDA-2 to be a PDA with two stacks (instead of one). Assume that the stacks can be manipulated independently and that the machine accepts iff it is in an accepting state and both stacks are empty when it runs out of input. Describe the operation of a PDA-2 that accepts $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$. (Note: we will see, in Section 17.5.2, that the PDA-2 is equivalent to the Turing machine in the sense that any language that can be accepted by one can be accepted by the other.)

M will have three states. In the first, it has seen only `a`'s. In the second, it has seen zero or more `a`'s, followed by one or more `b`'s. In the third, it has seen zero or more `a`'s, one or more `b`'s, and one or more `c`'s. In state 1, each time it sees an `a`, it will push it onto both stacks. In state 2, it will pop one `a` for each `b` it sees. In state 3, it will pop one `a` for each `c` it sees. It will accept if both stacks are empty when it runs out of input.

4 Some Important Ideas Before We Start

1) Describe in clear English or pseudocode a decision procedure to answer the question: given a list of integers N and an individual integer n, is there any element of N that is a factor of n?

decidefactor(n: integer, N: list of integers) =
  For each element i of N do:
    If i is a factor of n, then halt and return *True*.
  Halt and return *False*.

2) Given a Java program p and the input 0, consider the question, "Does p ever output anything?"
   a) Describe a semidecision procedure that answers this question.

Run p on 0. If it ever outputs something, halt and return *True*.

   b) Is there an obvious way to turn your answer to part (a) into a decision procedure?

No and there isn't any other way to create a decision procedure either.

3) Let $L = \{w \in \{\texttt{a}, \texttt{b}\}^* : w = w^R\}$. What is *chop*($L$)?

chop(L) = $\{w \in \{\texttt{a}, \texttt{b}\}^* : w = w^R$ and $|w|$ is even$\}$.

4) Are the following sets closed under the following operations? Prove your answer. If a set is not closed under the operation, what is its closure under the operation?

   a) $L = \{w \in \{\texttt{a}, \texttt{b}\}^* : w$ ends in $\texttt{a}\}$ under the function *odds*, defined on strings as follows: *odds*(s) = the string that is formed by concatenating together all of the odd numbered characters of s. (Start numbering the characters at 1.) For example, *odds*(ababbbb) = aabb.

Not closed. If $|w|$ is even, then the last character of *odds*(w) will be the next to the last character of w, which can be either a or b. For any w, $|odds(w)| \leq |w|$, and the shortest string in L has length 1. So the closure is $\{\texttt{a}, \texttt{b}\}^+$.

   b) FIN (the set of finite languages) under the function *oddsL*, defined on languages as follows:
      $oddsL(L) = \{w : \exists x \in L\ (w = odds(x))\}$

FIN is closed under the function *OddsL*. Each string in L can contribute at most one string to *OddsL*(L). So $|OddsL(L)| \leq |L|$.

   c) INF (the set of infinite languages) under the function *oddsL*.

INF is closed under the function *OddsL*. If INF were not closed under *OddsL*, then there would be some language L such that $|L|$ is infinite but $|OddsL(L)|$ were finite. If $|OddsL(L)|$ is finite, then there is a longest string in it. Let the length of one such longest string be n. If $|L|$ is infinite, then there is no longest string in L. So there must be some string w in L of length at least $2n + 2$. That string w will cause a string of length at least $n + 1$ to be in *OddsL*(L). But if $|OddsL(L)|$ is finite, the length of its longest string is n. Contradiction.

   d) FIN under the function *maxstring*, defined in Example 8.22**Error! Reference source not found.**.

FIN is closed under the function *maxstring*. Each string in L can contribute at most one string to *maxstring*(L). So $|maxstring(L)| \leq |L|$.

e) INF under the function *maxstring*.

INF is not closed under *maxstring*. a* is infinite, but *maxstring*(a*) = ∅, which is finite.

5) Let Σ = {a, b, c}. Let *S* be the set of all languages over Σ. Let *f* be a binary function defined as follows:

$$f: S \times S \rightarrow S$$
$$f(x, y) = x - y$$

Answer each of the following questions and defend your answers:

a) Is *f* one-to-one?

No, *f* is not one-to-one. Counterexample: {a, b, c} - {b, c} = {a} and {a, b} - {b} = {a}.

b) Is *f* onto?

Yes, *f* is onto. For any language *L*, *L* - ∅ = *L*.

c) Is *f* commutative?

No. *f* is not commutative. Counterexample: {a, b} - {b} = {a}, but {b} - {a, b} = ∅.

6) Describe a program, using *choose*, to:

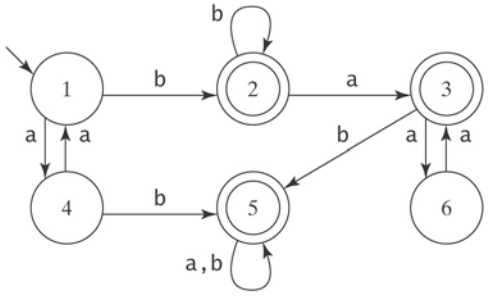
a) Play Sudoku 💻.

b) Solve Rubik's Cube® 💻.

Part II: Regular Languages

5 Finite State Machines

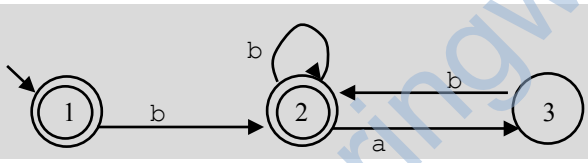
1) Give a clear English description of the language accepted by the following FSM:



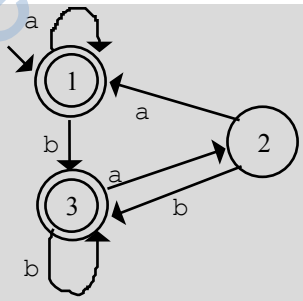
All strings of a's and b's consisting of an even number of a's, followed by at least one b, followed by zero or an odd number of a's.

2) Build a deterministic FSM for each of the following languages:

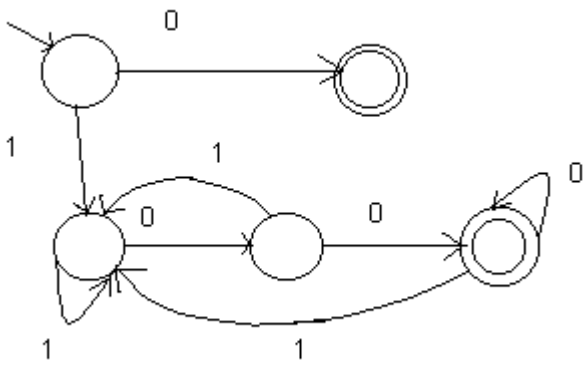
a) $\{w \in \{a, b\}^* : \text{every a in } w \text{ is immediately preceded and followed by b}\}$.



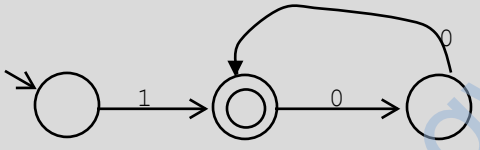
b) $\{w \in \{a, b\}^* : w \text{ does not end in ba}\}$.



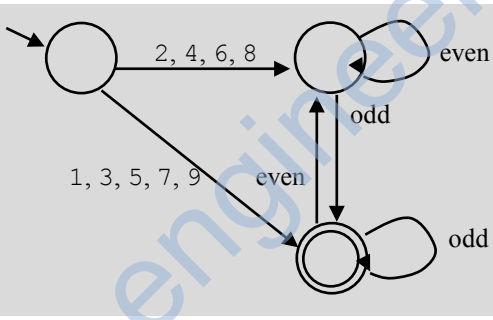
c) {$w \in \{0, 1\}^*$: w corresponds to the binary encoding, without leading 0's, of natural numbers that are evenly divisible by 4}.



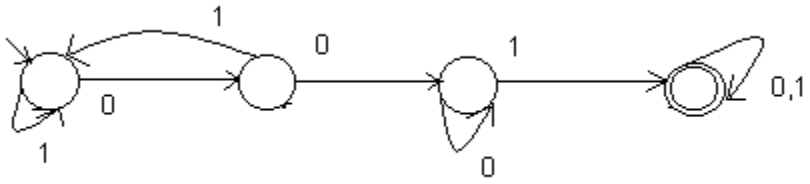
d) {$w \in \{0, 1\}^*$: w corresponds to the binary encoding, without leading 0's, of natural numbers that are powers of 4}.



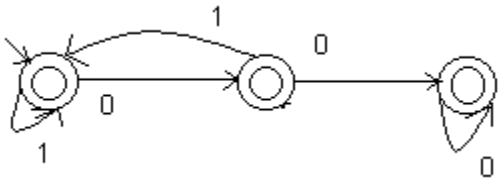
e) {$w \in \{0\text{-}9\}^*$: w corresponds to the decimal encoding, without leading 0's, of an odd natural number}.



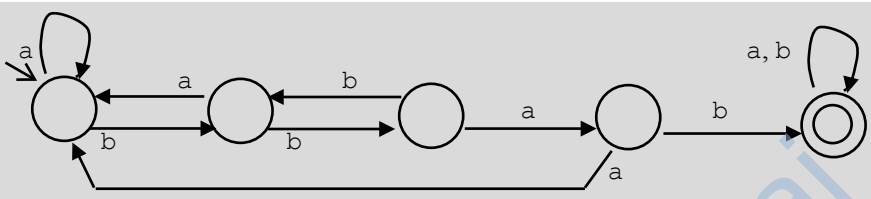
f) {$w \in \{0, 1\}^*$: w has 001 as a substring}.



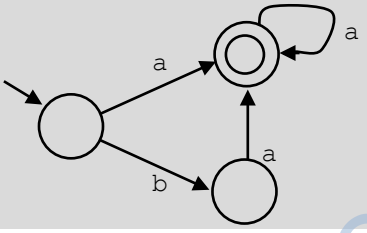
g) $\{w \in \{0, 1\}^* : w$ does not have 001 as a substring$\}$.



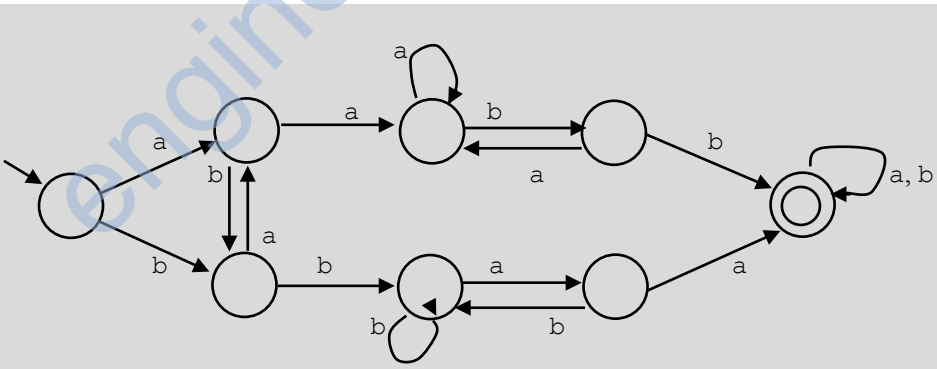
h) $\{w \in \{a, b\}^* : w$ has bbab as a substring$\}$.



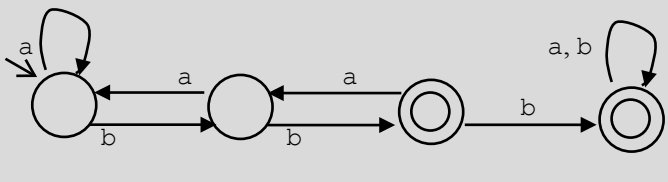
i) $\{w \in \{a, b\}^* : w$ has neither ab nor bb as a substring$\}$.



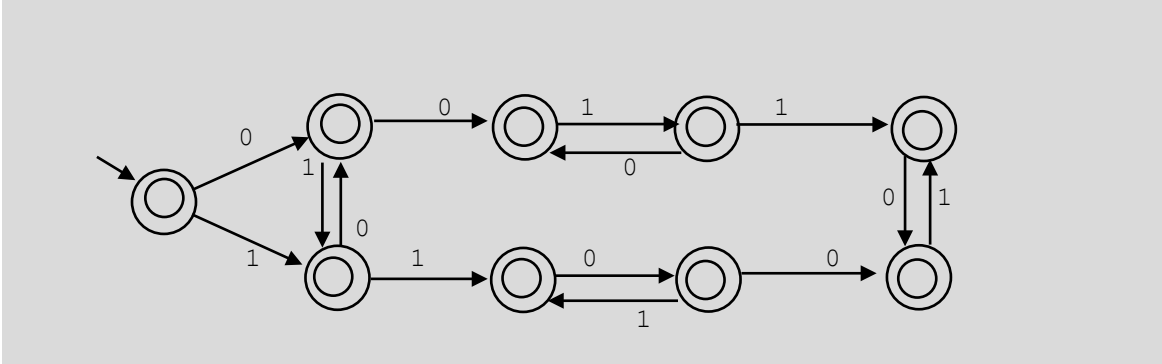
j) $\{w \in \{a, b\}^* : w$ has both aa and bb as a substrings$\}$.



k) $\{w \in \{a, b\}^* : w$ contains at least two b's that are not immediately followed by a's$\}$.

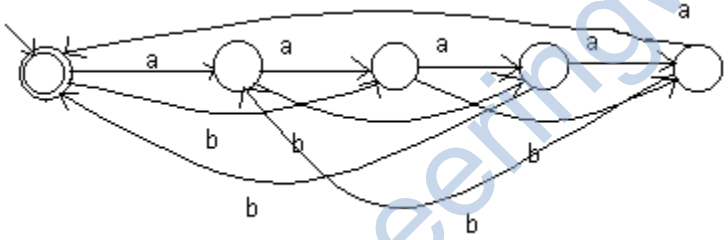


l) The set of binary strings with at most one pair of consecutive `0`'s and at most one pair of consecutive `1`'s.



m) $\{w \in \{0, 1\}^* :$ none of the prefixes of w ends in $0\}$.

n) $\{w \in \{a, b\}^* : (\#_a(w) + 2 \cdot \#_b(w)) \equiv_5 0\}$. ($\#_a w$ is the number of `a`'s in w).



3) Consider the children's game Rock, Paper, Scissors 💻. We'll say that the first player to win two rounds wins the game. Call the two players *A* and *B*.

a) Define an alphabet Σ and describe a technique for encoding Rock, Paper, Scissors games as strings over Σ. (Hint: each symbol in Σ should correspond to an ordered pair that describes the simultaneous actions of *A* and *B*.)

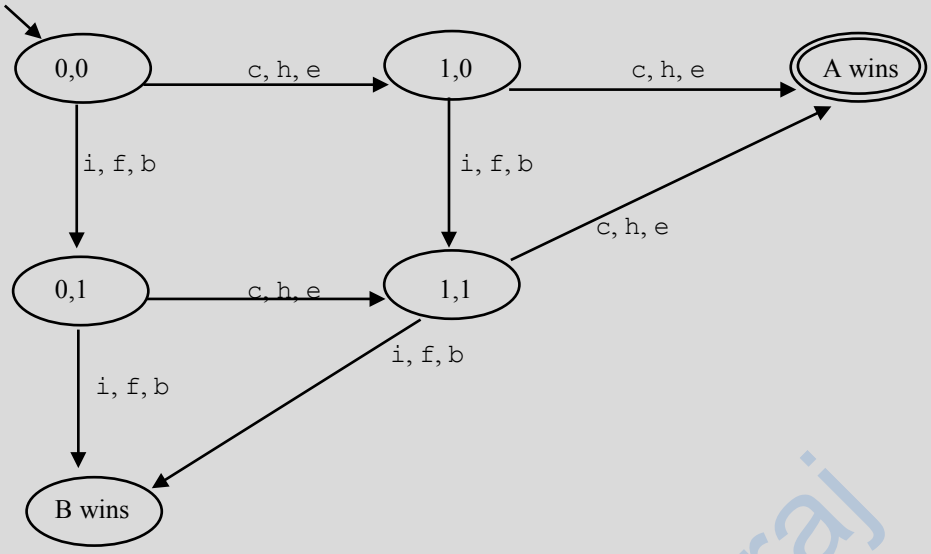
Let Σ have 9 characters. We'll use the symbols `a` - `i` to correspond to the following events. Let the first element of each pair be A's move. The second element of each pair will be B's move.

a	b	c	d	e	f	g	h	i
R, R	R, P	R, S	P, P	P, R	P, S	S, S	S, P	S, R

A Rock, Paper, Scissors game is a string of the symbols `a` - `i`. We'll allow strings of arbitrary length, but once one player as won two turns, no further events affect the outcome of the match.

b) Let L_{RPS} be the language of Rock, Paper, Scissors games, encoded as strings as described in part (a), that correspond to wins for player *A*. Show a DFSM that accepts L_{RPS}.

In the following diagram, a state with the name (*n*, *m*) corresponds to the case where player *A* has won *n* games and player *B* has one *m* games.

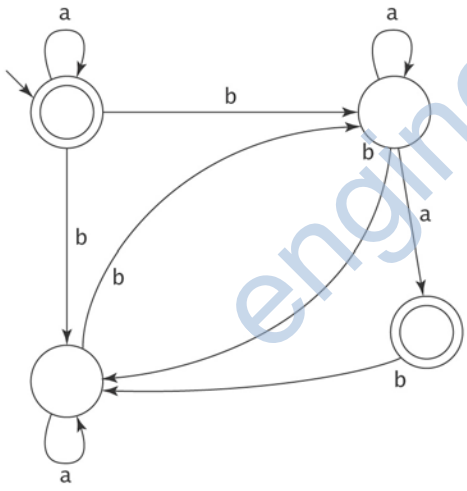


In addition, from every state, there is a transition back to itself labeled `a`, `d`, `g` (since the match status is unchanged if both players make the same move). And, from the two winning states, there is a transition back to that same state with all other labels (since, once someone has won, future events don't matter).

4) If M is a DFSM and $\varepsilon \in L(M)$, what simple property must be true of M?

The start state of M must be an accepting state.

5) Consider the following NDFSM M:

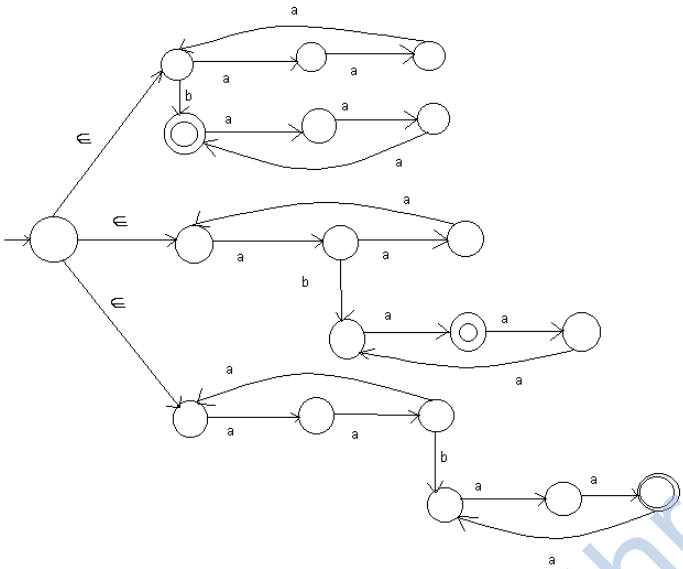


For each of the following strings w, determine whether $w \in L(M)$:

a) `aabbba`. Yes.
b) `bab`. No.
c) `baba`. Yes.

6) Show a possibly nondeterministic FSM to accept each of the following languages:

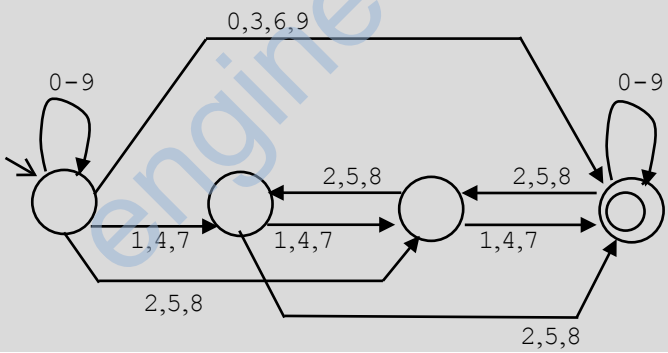
a) $\{a^n ba^m : n, m \geq 0, n \equiv_3 m\}$.



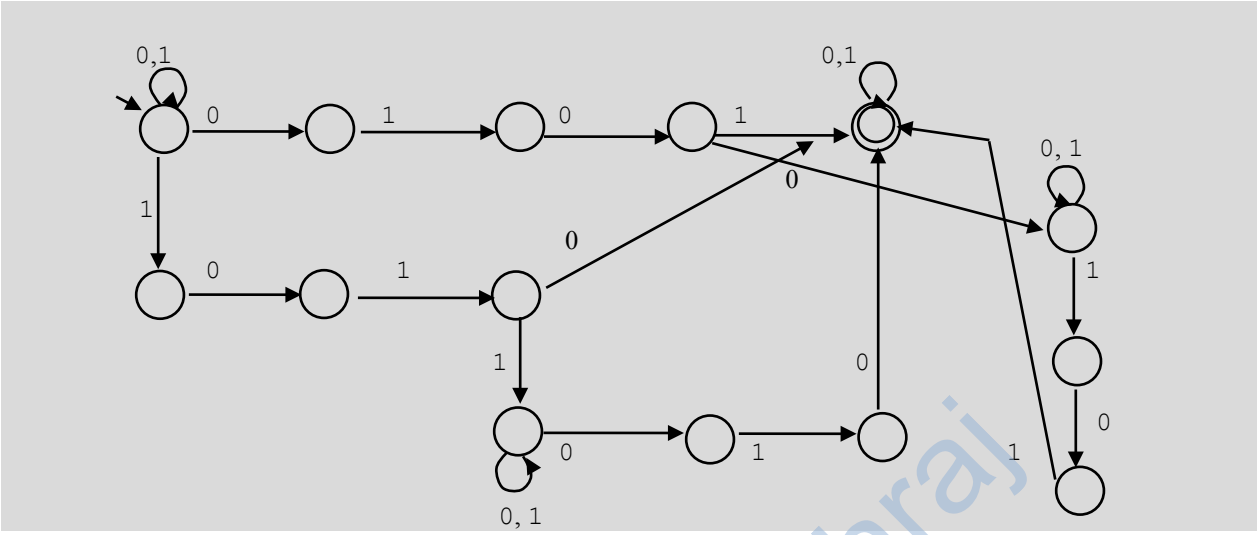
b) $\{w \in \{a, b\}^* : w$ contains at least one instance of aaba, bbb or ababa$\}$.

c) $L = \{w \in \{0\text{-}9\}^* : w$ represents the decimal encoding of a natural number whose encoding contains, as a substring, the encoding of a natural number that is divisible by 3$\}$.

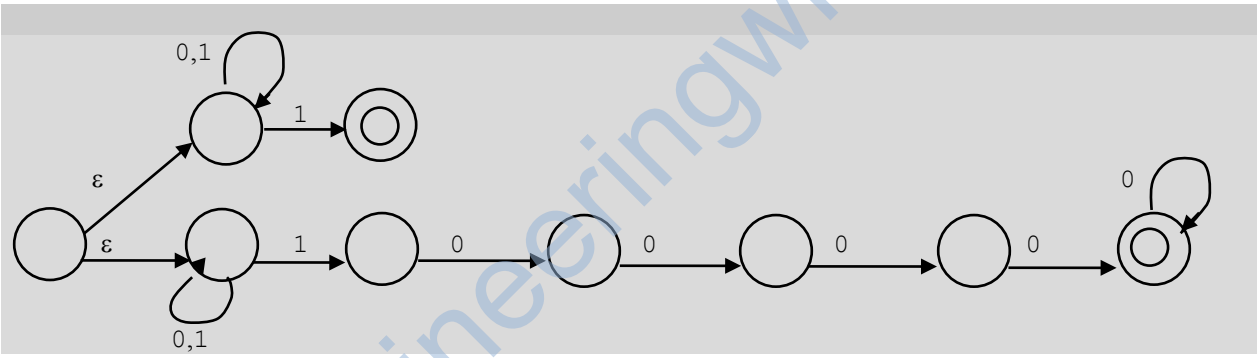
Note that 0 is a natural number that is divisible by 3. So any string that contains even one 0, 3, 6, or 9 is in L, no matter what else it contains. Otherwise, to be in L, there must be a sequence of digits whose sum equals 0 mod 3.



d) {*w* ∈ {0, 1}* : *w* contains both 101 and 010 as substrings}.



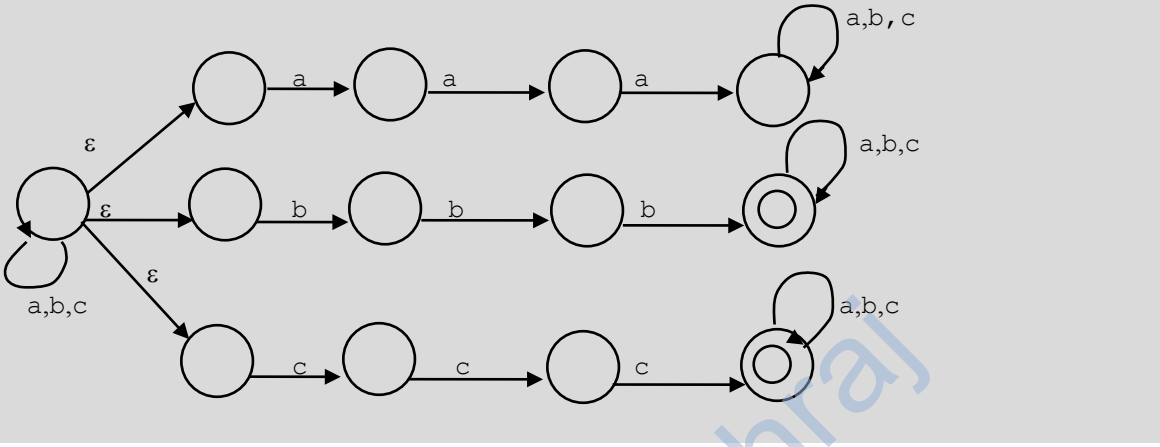
e) {*w* ∈ {0, 1}* : *w* corresponds to the binary encoding of a positive integer that is divisible by 16 or is odd}.



f) {*w* ∈ {a, b, c, d, e}* : |*w*| ≥ 2 and *w* begins and ends with the same symbol}.

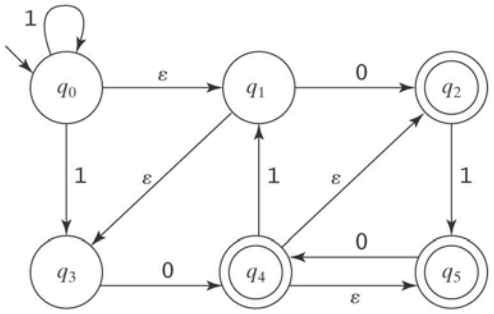
Guess which of the five symbols it is. Go to a state for each. Then, from each such state, guess that the next symbol is not the last and guess that it is.

7) Show an FSM (deterministic or nondeterministic) that accepts $L = \{w \in \{a, b, c\}^* : w$ contains at least one substring that consists of three identical symbols in a row$\}$. For example:
   - The following strings are in L: aabbb, baacccbbb.
   - The following strings are not in L: ε, aba, abababab, abcbcab.

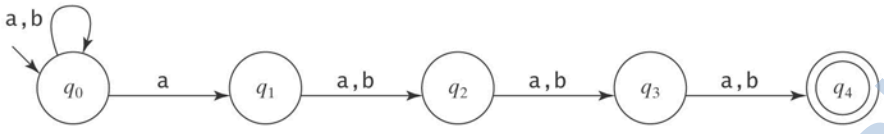


8) Show a deterministic FSM to accept each of the following languages. The point of this exercise is to see how much harder it is to build a deterministic FSM for tasks like these than it is to build an NDFSM. So do not simply built an NDFSM and then convert it. But do, after you build a DFSM, build an equivalent NDFSM.
   a) $\{w \in \{a, b\}^* :$ the fourth from the last character is a$\}$.
   b) $\{w \in \{a, b\}^* : \exists x, y \in \{a, b\}^* : ((w = x\,\mathtt{abbaa}\,y) \vee (w = x\,\mathtt{baba}\,y))\}$.

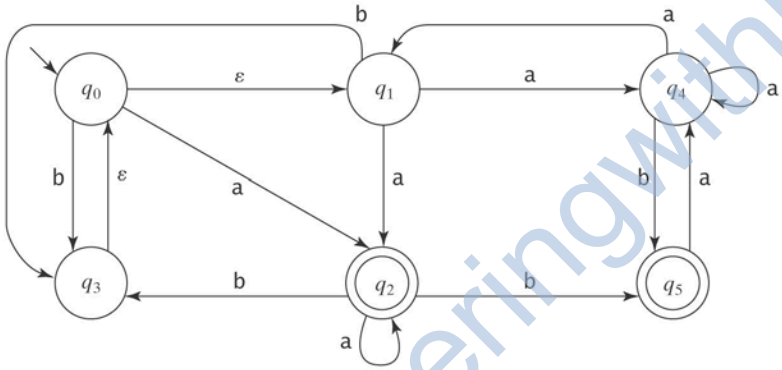
9) For each of the following NDFSMs, use *ndfsmtodfsm* to construct an equivalent DFSM. Begin by showing the value of *eps*(q) for each state q:



(a)



(b)

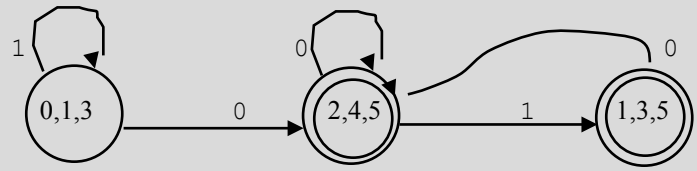


(c)

a)

s	*eps*(*s*)
q_0	$\{q_0, q_1, q_3\}$
q_1	$\{q_1, q_3\}$
q_2	$\{q_2\}$
q_3	$\{q_3\}$
q_4	$\{q_2, q_4, q_5\}$
q_5	$\{q_5\}$

$\{q_0, q_1, q_3\}$	0	$\{q_2, q_4, q_5\}$
	1	$\{q_0, q_1, q_3\}$
$\{q_2, q_4, q_5\}$	0	$\{q_2, q_4, q_5\}$
	1	$\{q_1, q_3, q_5\}$
$\{q_1, q_3, q_5\}$	0	$\{q_2, q_4, q_5\}$
	1	{ }



b)

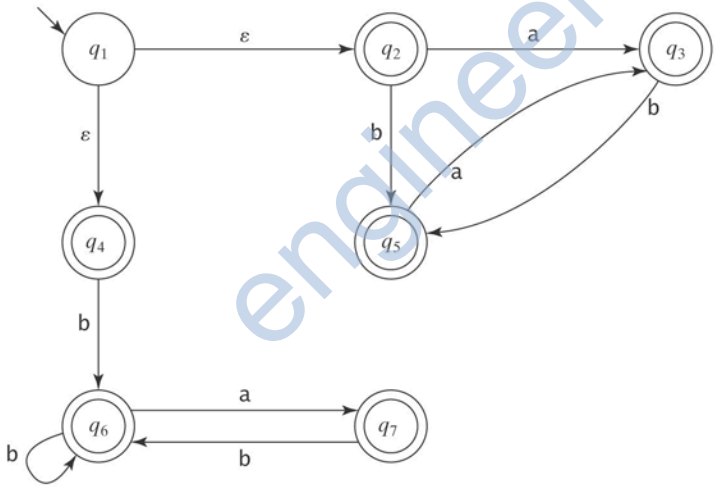
c)

s	***eps*(*s*)**
q_0	$\{q_0, q_1\}$
q_1	$\{q_1\}$
q_2	$\{q_2\}$
q_3	$\{q_3, q_0, q_1\}$
q_4	$\{q_4\}$
q_5	$\{q_5\}$

$\{q_0, q_1\}$	a	$\{q_2, q_4\}$
	b	$\{q_0, q_1, q_3\}$
$\{q_2, q_4\}$	a	$\{q_1, q_2, q_4\}$
	b	$\{q_0, q_1, q_3, q_5\}$
$\{q_0, q_1, q_3\}$	a	$\{q_2, q_4,\}$
	b	$\{q_0, q_1, q_3\}$
$\{q_1, q_2, q_4\}$	a	$\{q_1, q_2, q_4\}$
	b	$\{q_0, q_1, q_3, q_5\}$
$\{q_0, q_1, q_3, q_5\}$	a	$\{q_2, q_4\}$
	b	$\{q_0, q_1, q_3\}$

Accepting state is $\{q_0, q_1, q_3, q_5\}$. 1

10) Let *M* be the following NDFSM. Construct (using *ndfsmtodfsm*), a DFSM that accepts ¬*L*(*M*).



1) Complete by creating Dead state *D* and adding the transitions: {3, a, *D*}, {5, b, *D*}, {4, a, *D*}, {7, a, *D*}, {*D*, a, *D*}, {*D*, b, *D*}.

2) Convert to deterministic:

eps{1} = {1, 2, 4}

{1, 2, 4}, a, {3, *D*}
{1, 2, 4}, b, {5, 6}
{3, *D*}, a, {*D*}
{3, *D*}, b {5, *D*}
{5, 6}, a, {3, 7}
{5, 6}, b, {*D*, 6}
{*D*}, a, {*D*}
{*D*}, b, {*D*}
{5, *D*}, a, {3, *D*}
{5, *D*}, b, {*D*}
{3, 7}, a, {*D*}
{3, 7}, b, {5, 6}
{*D*, 6}, a, {*D*, 7}
{*D*, 6}, b, {*D*, 6}
{*D*, 7}, a , {*D*}
{*D*, 7}, b, {*D*, 6}
All these states are accepting except {*D*}.

3) Swap accepting and nonnonaccepting states, making all states nonaccepting except {*D*).

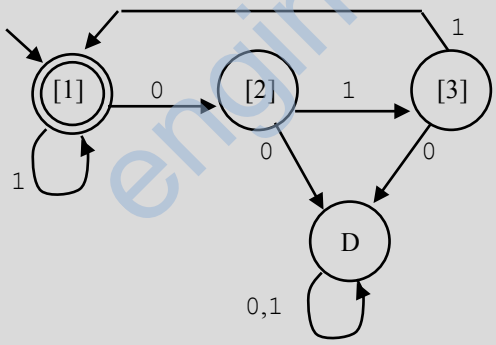
11) For each of the following languages *L*:

(i) Describe the equivalence classes of $\approx_L$.

(ii) If the number of equivalence classes of $\approx_L$ is finite, construct the minimal DFSM that accepts *L*.

a) {*w* ∈ {0, 1}* : every 0 in *w* is immediately followed by the string 11}.

[1] {in *L*}
[2] {otherwise in *L* except ends in 0}
[3] {otherwise in *L* except ends in 01}
[D] {corresponds to the Dead state: string contains at least one instance of 00 or 010}



b) {*w* ∈ {0, 1}* : *w* has either an odd number of 1's and an odd number of 0's or it has an even number of 1's and an even number of 0's}.

c) {*w* ∈ {a, b}* : *w* contains at least one occurrence of the string aababa}.

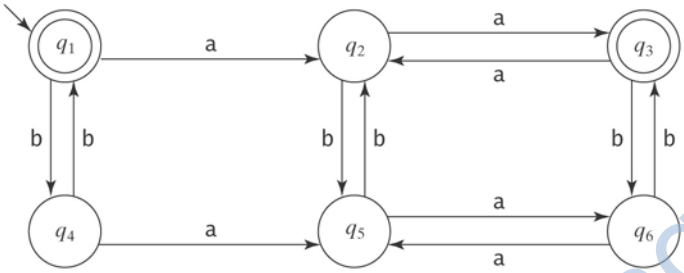
d) $\{ww^R : w \in \{a, b\}^*\}$.

[1] {ε} in L
[2] {a}
[3] {b}
[4] {aa} in L
[5] {ab}

And so forth. Every string is in a different equivalence class because each could become in L if followed by the reverse of itself but not if followed by most other strings. This language is not regular.

e) $\{w \in \{a, b\}^* : w$ contains at least one a and ends in at least two b's$\}$.
f) $\{w \in \{0, 1\}^* :$ there is no occurrence of the substring 000 in $w\}$.

12) Let M be the following DFSM. Use *minDFSM* to minimize M.



Initially, *classes* = {[1, 3], [2, 4, 5, 6]}.

At step 1:

((1, a), [2, 4, 5, 6])	((3, a), [2, 4, 5, 6])		No splitting required here.
((1, b), [2, 4, 5, 6])	((3, b), [2, 4, 5, 6])		
((2, a), [1, 3])	((4, a), [2, 4, 5, 6])	((5, a), [2, 4, 5, 6])	((6, a), [2, 4, 5, 6])
((2, b), [2, 4, 5, 6])	((4, b), [1, 3])	((5, b), [2, 4, 5, 6])	((6, b), [1, 3])

These split into three groups: [2], [4, 6], and [5]. So classes is now {[1, 3], [2], [4, 6], [5]}.

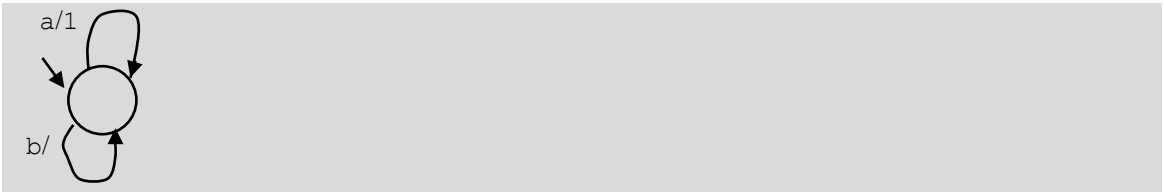
At step 2, we must consider [4, 6]:

((4, a), [5])	((6, a), [5])
((4, b), [1])	((6, b), [1])

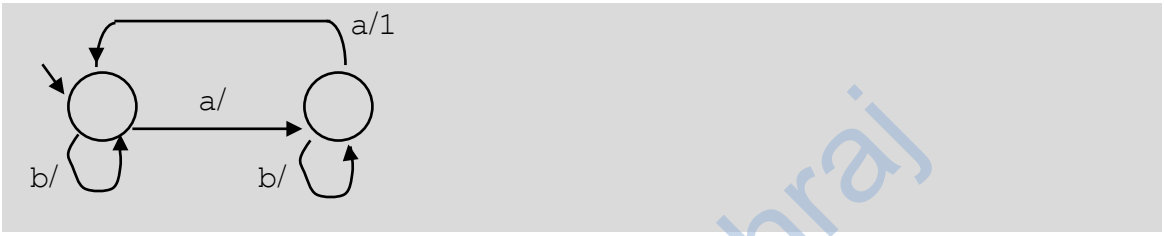
No further splitting is required. The minimal machine has the states: {[1, 3], [2], [4, 6], [5]}, with transitions as shown above.

13) Construct a deterministic finite state transducer with input {a, b} for each of the following tasks:

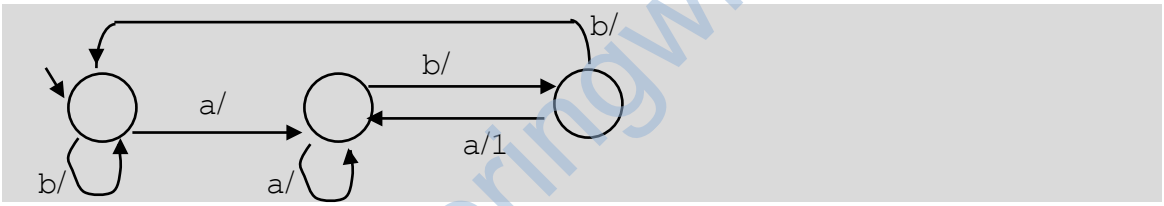
a) On input *w*, produce 1^n, where $n = \#_a(w)$.



b) On input *w*, produce 1^n, where $n = \#_a(w)/2$.



c) On input *w*, produce 1^n, where *n* is the number of occurrences of the substring aba in *w*.



14) Construct a deterministic finite state transducer that could serve as the controller for an elevator. Clearly describe the input and output alphabets, as well as the states and the transitions between them.

15) Consider the problem of counting the number of words in a text file that may contain letters plus any of the following characters:

<*blank*> <*linefeed*> <*end-of-file*> , . ; : ? !

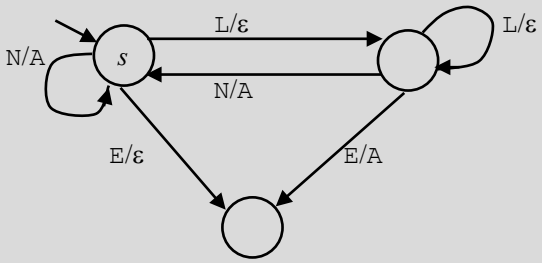
Define a word to be a string of letters that is preceded by either the beginning of the file or some non-letter character and that is followed by some non-letter character. For example, there are 11 words in the following text:

The <*blank*> <*blank*> cat <*blank*> <*linefeed*>
saw <*blank*> the <*blank*> <*blank*> <*blank*> rat <*linefeed*>
<*blank*> with
<*linefeed*> a <*blank*> hat <*linefeed*>
on <*blank*> the <*blank*> <*blank*> mat <*end-of-file*>

Describe a very simple finite-state transducer that reads the characters in the file one at a time and solves the word-counting problem. Assume that there exists an output symbol with the property that, every time it is generated, an external counter gets incremented.

We'll let the input alphabet include the symbols: L (for a letter), N (for a nonletter), and E (for end-of-file). The output alphabet will contain just a single symbol A (for add one to the counter). We don't need any accepting states.

Let $M = \{K, \{L, N, E\}, \{A\}, s, \varnothing, \delta, D\}$, where K, δ, and D are as follows.



16) Real traffic light controllers are more complex than the one that we drew in Example 5.29.
   a) Consider an intersection of two roads controlled by a set of four lights (one in each direction). Don't worry about allowing for a special left-turn signal. Design a controller for this four-light system.
   b) As an emergency vehicle approaches an intersection, it should be able to send a signal that will cause the light in its direction to turn green and the light in the cross direction to turn yellow and then red. Modify your design to allow this.

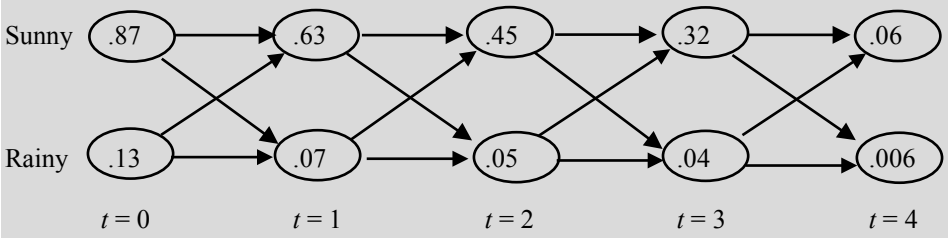
17) Real bar code systems are more complex than the one we sketched in the book. They must be able to encode all ten digits, for example. There are several industry-standard formats for bar codes, including the common UPC code found on nearly everything we buy. Search the web. Find the encoding scheme used for UPC codes. Describe a finite state transducer that reads the bars and outputs the corresponding decimal number.

18) Extend the description of the Soundex FSM that was started in Example 5.33 so that it can assign a code to the name Pfifer. Remember that you must take into account the fact that every Soundex code is made up of exactly four characters.

19) Consider the weather/passport HMM of Example 5.37. Trace the execution of the Viterbi and forward algorithms to answer the following questions:

Students can solve these problems by hand, by writing code, or by using standard packages. These solutions were created using the HMM package in Matlab.

   a) Suppose that the report ###L is received from Athens. What was the most likely weather during the time of the report?

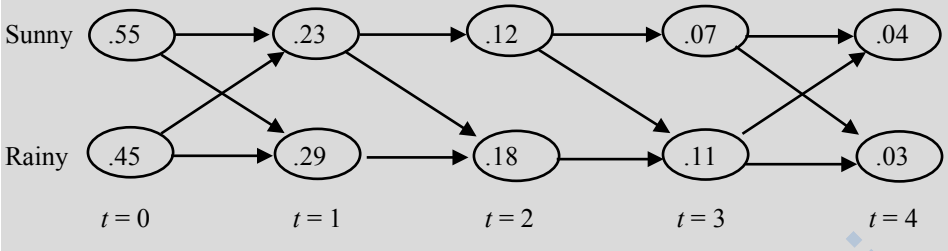


So the most likely series of weather reports is Sunny, Sunny, Sunny, Sunny.

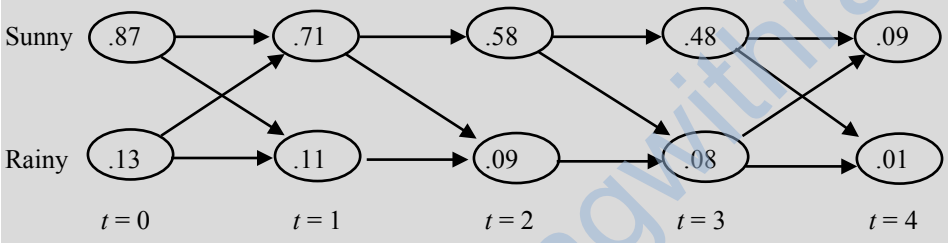
b) Is it more likely that ###L came from London or from Athens?

To solve this problem, we run the forward algorithm on both the London and the Athens model and see which has the higher probability of outputting the observed sequence. So we have:

London:



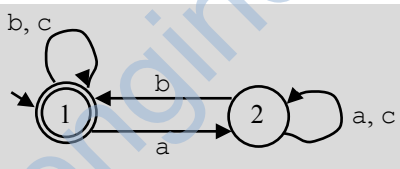
Athens:



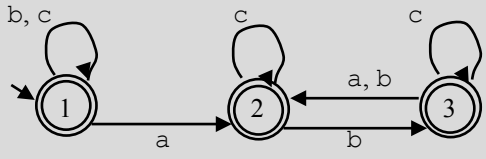
The total probability for London is thus .07. The total probability for Athens is .1. So Athens is more likely to have produced the output ###L.

20) Construct a Büchi automaton to accept each of the following languages of infinite length strings:

a) $\{w \in \{a, b, c\}^{\omega}$: after any occurrence of an a there is eventually an occurrence of a b$\}$.

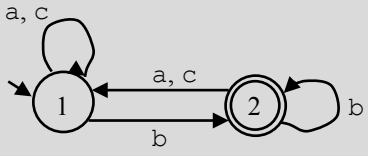


b) $\{w \in \{a, b, c\}^{\omega}$: between any two a's there is an odd number of b's$\}$.



We have omitted the dead state, which is reached from state 2 on an a.

c) $\{w \in \{a, b, c\}^{\omega}$: there never comes a time after which no b's occur$\}$.



21) In H.2, we describe the use of statecharts as a tool for building complex systems. A statechart is a hierarchically structured transition network model. Statecharts aren't the only tools that exploit this idea. Another is Simulink® 💻, which is one component of the larger programming environment Matlab® 💻. Use Simulink to build an FSM simulator.

22) In I.1.2, we describe the Alternating Bit Protocol for handling message transmission in a network. Use the FSM that describes the sender to answer the question, "Is there any upper bound on the number of times a message may be retransmitted?"

No. The loop from either of the need ACK states to the corresponding timeout state can happen any number of times.

23) In J.1, we show an FSM model of simple intrusion detection device that could be part of a building security system. Extend the model to allow the system to have two zones that can be armed and disarmed independently of each other.

6 Regular Expressions

1) Describe in English, as briefly as possible, the language defined by each of these regular expressions:
   a) (b ∪ ba) (b ∪ a)* (ab ∪ b).

      The set of strings over the alphabet {a, b} that start and end with b.

   b) (((a*b*)*ab) ∪ ((a*b*)*ba))(b ∪ a)*.

      The set of strings over the alphabet {a, b} that contain at least one occurrence of ab or ba.

2) Write a regular expression to describe each of the following languages:
   a) {*w* ∈ {a, b}* : every a in *w* is immediately preceded and followed by b}.

      (b ∪ bab)*

   b) {*w* ∈ {a, b}* : *w* does not end in ba}.

      ε ∪ a ∪ (a ∪ b)* (b ∪ aa)

   c) {*w* ∈ {0, 1}* : ∃*y* ∈ {0, 1}* (|*xy*| is even)}.

      (0 ∪ 1)*

   d) {*w* ∈ {0, 1}* : *w* corresponds to the binary encoding, without leading 0's, of natural numbers that are evenly divisible by 4}.

      (1(0 ∪ 1)* 00) ∪ 0

   e) {*w* ∈ {0, 1}* : *w* corresponds to the binary encoding, without leading 0's, of natural numbers that are powers of 4}.

      1(00)*

   f) {*w* ∈ {0-9}* : *w* corresponds to the decimal encoding, without leading 0's, of an odd natural number}.

      (ε ∪ ((1-9)(0-9)*))(1 ∪ 3 ∪ 5 ∪ 7 ∪ 9)

   g) {*w* ∈ {0, 1}* : *w* has 001 as a substring}.

      (0 ∪ 1)* 001 (0 ∪ 1)*

   h) {*w* ∈ {0, 1}* : *w* does not have 001 as a substring}.

      (1 ∪ 01)* 0*

   i) {*w* ∈ {a, b}* : *w* has bba as a substring}.

      (a ∪ b)* bba (a ∪ b)*

j) {$w \in$ {a, b}* : w has both aa and bb as substrings}.

(a ∪ b)* aa (a ∪ b)* bb (a ∪ b)* ∪ (a ∪ b)* bb (a ∪ b)* aa (a ∪ b)*

k) {$w \in$ {a, b}* : w has both aa and aba as substrings}.

(a ∪ b)* aa (a ∪ b)* aba (a ∪ b)* ∪
(a ∪ b)* aba (a ∪ b)* aa (a ∪ b)* ∪
(a ∪ b)* aaba (a ∪ b)* ∪
(a ∪ b)* abaa (a ∪ b)*

l) {$w \in$ {a, b}* : w contains at least two b's that are not followed by an a}.

(a ∪ b)* bb ∪ (a ∪ b)* bbb (a ∪ b)*

m) {$w \in$ {0, 1}* : w has at most one pair of consecutive 0's and at most one pair of consecutive 1's}.

(ε ∪ 1)(01)*(ε ∪ 1) (01)* (ε ∪ (00 (ε ∪ (1 (01)* (ε ∪ 0))))) /* 11 comes first.
∪
(ε ∪ 0) (10)* (ε ∪ 0) (10)* (ε ∪ (11 (ε ∪ (0 (10)* (ε ∪ 1))))) /* 00 comes first.

n) {$w \in$ {0, 1}* : none of the prefixes of w ends in 0}.

1*

o) {$w \in$ {a, b}* : $\#_a(w) \equiv_3 0$}.

(b*ab*ab*a)*b*

p) {$w \in$ {a, b}* : $\#_a(w) \leq 3$}.

b* (a ∪ ε) b* (a ∪ ε) b* (a ∪ ε) b*

q) {$w \in$ {a, b}* : w contains exactly two occurrences of the substring aa}.

(b ∪ ab)*aaa(b ∪ ba)* ∪ (b ∪ ab)*aab(b ∪ ab)*aa(b ∪ ba)* (Either the two occurrences are contiguous, producing aaa, or they're not.)

r) {$w \in$ {a, b}* : w contains no more than two occurrences of the substring aa}.

(b ∪ ab)*(a ∪ ε) /* 0 occurrences of the substring aa
∪
(b ∪ ab)*aa(b ∪ ba)* /* 1 occurrence of the substring aa
∪
(b ∪ ab)*aaa(b ∪ ba)* ∪ (b ∪ ab)*aab(b ∪ ab)*aa(b ∪ ba)*
/* 2 occurrences of the substring aa

s) {$w \in$ {a, b}* - L}, where L = {$w \in$ {a, b}* : w contains bba as a substring}.

(a ∪ ba)* (ε ∪ b ∪ bbb*) = (a ∪ ba)*b*

t) $\{w \in \{0, 1\}^* :$ every odd length string in L begins with $11\}$.

$((0 \cup 1)(0 \cup 1))^* \cup 11(0 \cup 1)^*$

u) $\{w \in \{0\text{-}9\}^* :$ w represents the decimal encoding of an odd natural number without leading 0's.

$(\varepsilon \cup ((1\text{-}9)(0\text{-}9)^*))(1 \cup 3 \cup 5 \cup 7 \cup 9)$

v) $L_1 - L_2$, where $L_1 = a^*b^*c^*$ and $L_2 = c^*b^*a^*$.

$a^+b^+c^* \cup a^+b^*c^+ \cup a^*b^+c^+$ (The only strings that are in both L_1 and L_2 are strings composed of no more than one different letter. So those are the strings that we need to remove from L_1 to form the difference. What we're left with is strings that contain two or more distinct letters.)

w) The set of legal United States zipcodes 💻.

x) The set of strings that correspond to domestic telephone numbers in your country.

3) Simplify each of the following regular expressions:

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

$(a \cup b)^*$.

b) $(\varnothing^* \cup b) b^*$.

b^*.

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

$(a \cup b)^*$.

d) $((a \cup b)^*)^*$.

$(a \cup b)^*$.

e) $((a \cup b)^+)^*$.

$(a \cup b)^*$.

f) $a\,(\,(a \cup b)(b \cup a)\,)^* \cup a\,(\,(a \cup b)\,a\,)^* \cup a\,(\,(b \cup a)\,b\,)^*$.

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

4) For each of the following expressions E, answer the following three questions and prove your answer:

(i) Is E a regular expression?
(ii) If E is a regular expression, give a simpler regular expression.
(iii) Does E describe a regular language?

a) $((a \cup b) \cup (ab))^*$.

E is a regular expression. A simpler one is $(a \cup b)^*$. The language is regular.

b) $(a^+ a^n b^n)$.

E is not a regular expression. The language is not regular. It is $\{a^m b^n : m > n\}$.

c) $((ab)^* \varnothing)$.

E is a regular expression. A simpler one is ∅. The language is regular.

d) $(((ab) \cup c)^* \cap (b \cup c^*))$.

E is not a regular expression because it contains ∩. But it does describe a regular language (c^*) because the regular languages are closed under intersection.

e) $(\varnothing^* \cup (bb^*))$.

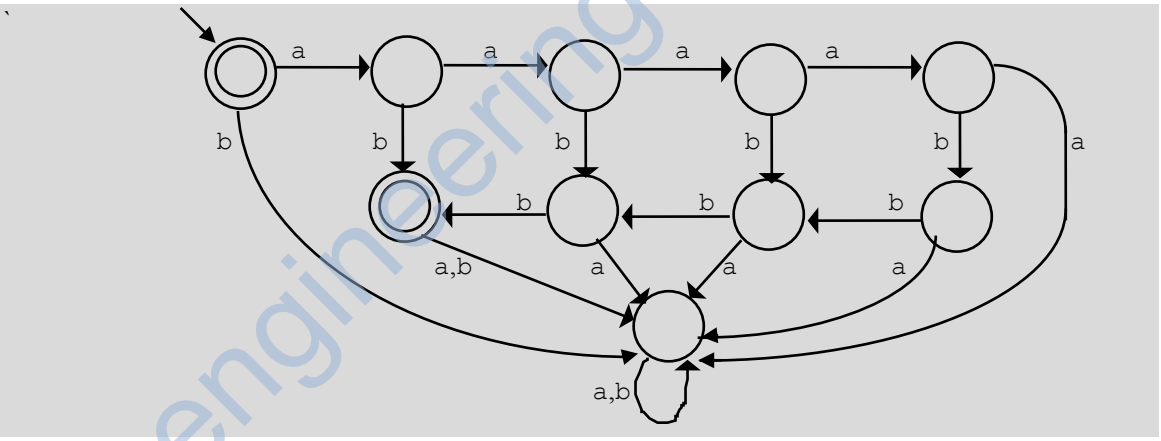
E is a regular expression. A (slightly) simpler one is $(\varepsilon \cup (bb^*))$. The language is regular.

5) Let $L = \{a^n b^n : 0 \le n \le 4\}$.

a) Show a regular expression for *L*.

$(\varepsilon \cup ab \cup aabb \cup aaabbb \cup aaaabbbb)$

b) Show an FSM that accepts *L*.

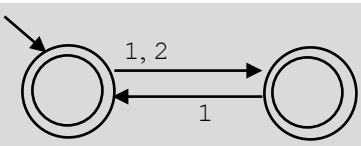


6) Let $L = \{w \in \{1, 2\}^* :$ for all prefixes *p* of *w*, if $|p| > 0$ and $|p|$ is even, then the last character of *p* is $1\}$.

a) Write a regular expression for *L*.

$((1 \cup 2)1)^* (1 \cup 2 \cup \varepsilon)$

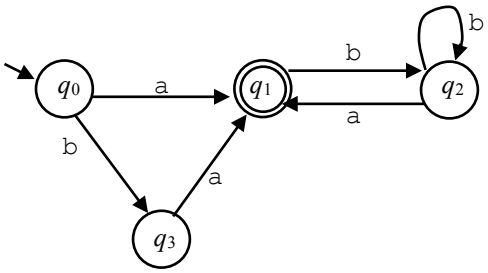
b) Show an FSM that accepts *L*.



7) Use the algorithm presented in the proof of Kleene's theorem to construct an FSM to accept the languages generated by the following regular expressions:

a) (b(b ∪ ε)b)*.
b) bab ∪ a*.

8) Let L be the language accepted by the following finite state machine:

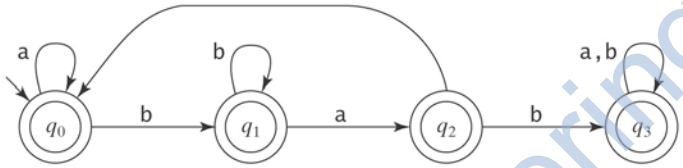


Indicate, for each of the following regular expressions, whether it correctly describes L:

a) (a ∪ ba)bb*a.
b) (ε ∪ b)a(bb*a)*.
c) ba∪ ab*a.
d) (a ∪ ba)(bb*a)*.

a) no; b) yes; c) no; d) yes.

9) Consider the following FSM M:



The first printing of the book has two mistakes in this figure: The transition from q_2 back to q_0 should be labeled a, and state q_3 should not be accepting. We'll give answers for the printed version (in which we'll simply ignore the unlabeled transition from q_2 to q_0) and the correct version.

a) Show a regular expression for $L(M)$.

Printed version: a* ∪ a*bb* ∪ a*bb*a ∪ a*bb*ab (a ∪ b)*
Correct version: (a ∪ bb*aa)* (ε ∪ bb*(a ∪ ε)).

b) Describe $L(M)$ in English.

Printed version: No obvious concise description.
Correct version: All strings in {a, b}* that contain no occurrence of bab.

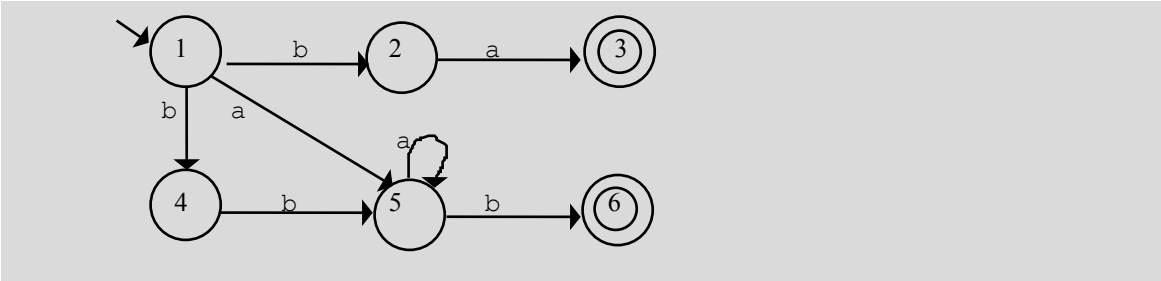
10) Consider the FSM M of Example 5.3. Use *fsmtoregexheuristic* to construct a regular expression that describes $L(M)$.

11) Consider the FSM M of Example 6.9. Apply *fsmtoregex* to M and show the regular expression that results.

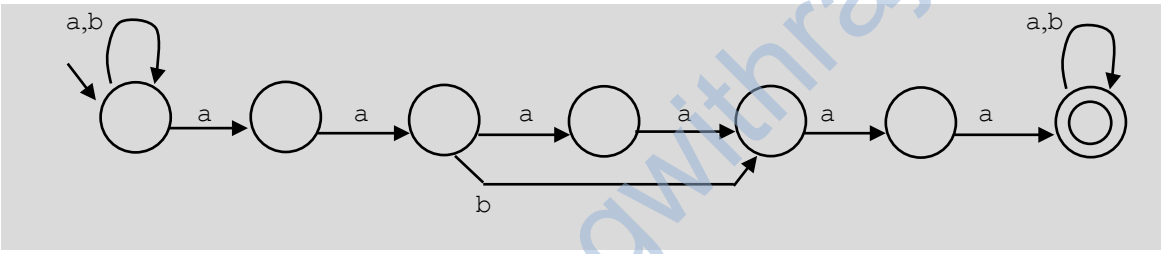
12) Consider the FSM M of Example 6.8. Apply *fsmtoregex* to M and show the regular expression that results. (Hint: this one is exceedingly tedious, but it can be done.)

13) Show a possibly nondeterministic FSM to accept the language defined by each of the following regular expressions:

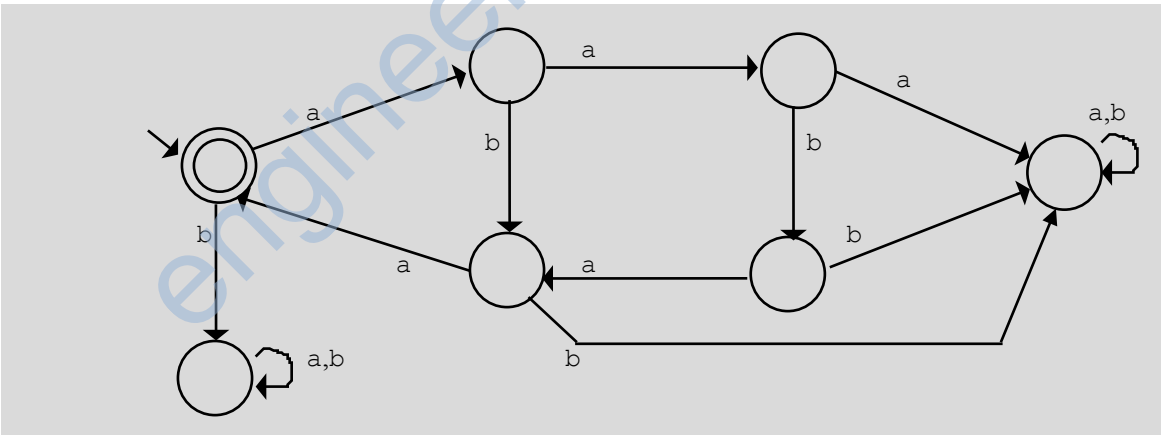
a) (((a ∪ ba) b ∪ aa)*.
b) (b ∪ ε)(ab)*(a ∪ ε).
c) (babb* ∪ a) *.
d) (ba ∪ ((a ∪ bb) a*b)).



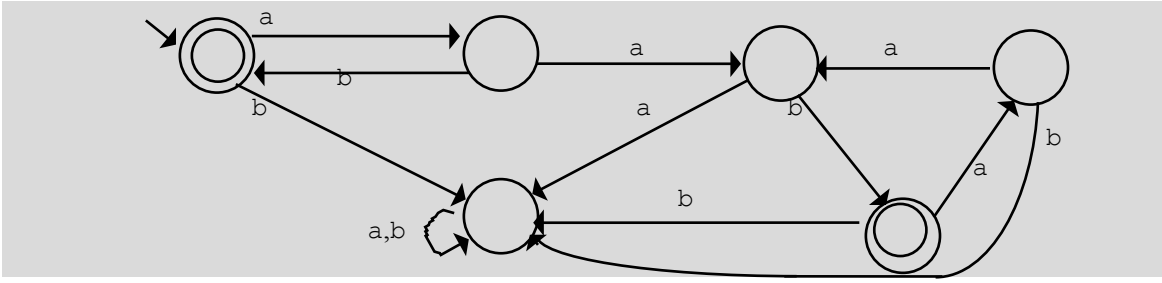
e) (a ∪ b)* aa (b ∪ aa) bb (a ∪ b)*.



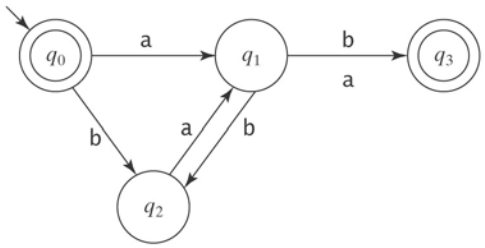
14) Show a DFSM to accept the language defined by each of the following regular expressions:
a) (aba ∪ aabaa)*



b) (ab)*(aab)*



15) Consider the following FSM *M*:



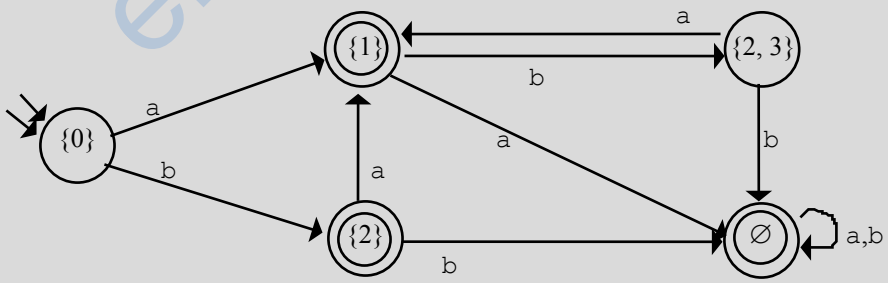
The first printing of the book has a mistake in this figure: There should not be an a labeling the transition from q_1 to q_3.

a) Write a regular expression that describes $L(M)$.

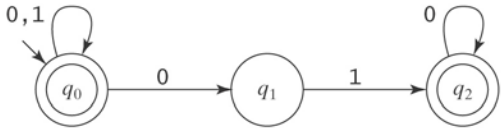
Printed version: ε ∪ ((a ∪ ba)(ba)*b ∪ a)
Correct version: ε ∪ ((a ∪ ba)(ba)*b).

b) Show a DFSM that accepts $\neg L(M)$.

This is for the correct version (without the extra label a):



16) Given the following DFSM M, write a regular expression that describes $\neg L(M)$:



Give a regular expression for $\neg L(M)$.

We first construct a deterministic FSM M^* equivalent to M. $M^* =$
$(\{\{q_0\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_1, q_2\}\}, \{0, 1\}, \delta^*, \{q_0\}, \{\{q_0, q_2\}, \{q_0, q_1, q_2\}\})$, where $\delta^* =$
$\{(\{q_0\}, 0, \{q_0, q_1\}),$
$(\{q_0\}, 1, \{q_0\}),$
$(\{q_0, q_1\}, 0, \{q_0, q_1\}),$
$(\{q_0, q_1\}, 1, \{q_0, q_2\}),$
$(\{q_0, q_2\}, 0, \{q_0, q_1, q_2\}),$
$(\{q_0, q_2\}, 1, \{q_0\}),$
$(\{q_0, q_1, q_2\}, 0, \{q_0, q_1, q_2\}),$
$(\{q_0, q_1, q_2\}, 1, \{q_0, q_2\})\}$

From M^*, we flip accepting and nonaccepting states to build M^{**}, which accepts $\neg L(M)$. $M^{**} = M^*$ except that $A^{**} = \{\}$.

So a regular expression that accepts $L(M^{**})$ is $\varnothing$.

17) Add the keyword `able` to the set in Example 6.13 and show the FSM that will be built by *buildkeywordFSM* from the expanded keyword set.

18) Let $\Sigma = \{a, b\}$. Let $L = \{\varepsilon, a, b\}$. Let R be a relation defined on Σ^* as follows: $\forall xy$ (xRy iff $y = xb$). Let R' be the reflexive, transitive closure of R. Let $L' = \{x : \exists y \in L\ (yR'x)\}$. Write a regular expression for L'.

$R' = \{(\varepsilon, \varepsilon), (\varepsilon, b), (\varepsilon, bb), (\varepsilon, bbb), \ldots (a, a), (a, ab), (a, abb), \ldots (b, b), (b, bb), (b, bbb), \ldots\}$.

So a regular expression for L' is: $(\varepsilon \cup a \cup b)b^*$.

19) In Appendix O, we summarize the main features of the regular expression language in Perl. What feature of that regular expression language makes it possible to write regular expressions that describe languages that aren't regular?

The ability to store strings of arbitrary length in variables and then require that those variables match later in the string.

20) For each of the following statements, state whether it is *True* or *False*. Prove your answer.

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

True.

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

True.

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

False.

d) $(a \cup b)^* b (a \cup b)^* \cup (a \cup b)^* a (a \cup b)^* = (a \cup b)^+$.

True.

e) $(a \cup b)^* b a (a \cup b)^* \cup a^* b^* = (a \cup b)^*$.

True.

f) $a^* b (a \cup b)^* = (a \cup b)^* b (a \cup b)^*$.

True.

g) If α and β are any two regular expressions, then $(\alpha \cup \beta)^* = \alpha(\beta\alpha \cup \alpha)$.

False.

h) If α and β are any two regular expressions, then $(\alpha\beta)^*\alpha = \alpha(\beta\alpha)^*$.

True.

7 Regular Gramamars

1) Show a regular grammar for each of the following languages:

a) $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ contains an odd number of a's and an odd number of b's$\}$.

$G = (\{EE, EO, OE, OO, \mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}, EE, R)$, where $R =$

$EE \rightarrow \mathtt{a}\ OE$
$EE \rightarrow \mathtt{b}\ EO$
$OE \rightarrow \mathtt{b}\ OO$
$OE \rightarrow \mathtt{a}\ EE$
$EO \rightarrow \varepsilon$
$EO \rightarrow \mathtt{a}\ OO$
$EO \rightarrow \mathtt{b}\ EE$
$OO \rightarrow \mathtt{a}\ EO$
$OO \rightarrow \mathtt{b}\ OE$

b) $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ does not end in aa$\}$.

$S \rightarrow \mathtt{a}A \mid \mathtt{b}B \mid \varepsilon$
$A \rightarrow \mathtt{a}C \mid \mathtt{b}B \mid \varepsilon$
$B \rightarrow \mathtt{a}A \mid \mathtt{b}B \mid \varepsilon$
$C \rightarrow \mathtt{a}C \mid \mathtt{b}B$

c) $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ contains the substring abb$\}$.

d) $\{w \in \{\mathtt{a}, \mathtt{b}\}^* :$ if w contains the substring aa then $|w|$ is odd$\}$.

It helps to begin by rewriting this as:

$\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ does not contain the substring aa or $|w|$ is odd$\}$

The easiest way to design this grammar is to build an FSM first. The FSM has seven states:

- S, the start state, is never reentered after the first character is read.
- T_1: No aa yet; even length; last character was a.
- T_2: No aa yet; odd length; last character was a.
- T_3: No aa yet; even length; last character was b.
- T_4: No aa yet; odd length; last character was b.
- T_5: aa seen; even length.
- T_6: aa seen; odd length.

Now we build a regular grammar whose nonterminals correspond to those states. So we have:

$S \rightarrow \mathtt{a}T_2 \mid \mathtt{b}T_4$
$T_1 \rightarrow \mathtt{a}T_6 \mid \mathtt{b}T_4 \mid \varepsilon$
$T_2 \rightarrow \mathtt{a}T_5 \mid \mathtt{b}T_3 \mid \varepsilon$
$T_3 \rightarrow \mathtt{a}T_2 \mid \mathtt{b}T_4 \mid \varepsilon$
$T_4 \rightarrow \mathtt{a}T_1 \mid \mathtt{b}T_3 \mid \varepsilon$
$T_5 \rightarrow \mathtt{a}T_6 \mid \mathtt{b}T_6$
$T_6 \rightarrow \mathtt{a}T_5 \mid \mathtt{b}T_5 \mid \varepsilon$

e) $\{w \in \{a, b\}^* : w$ does not contain the substring $aabb\}$.

$S \rightarrow aA$	$A \rightarrow aB$	$B \rightarrow aB$	$C \rightarrow aA$
$S \rightarrow bS$	$A \rightarrow bS$	$B \rightarrow bC$	$C \rightarrow \varepsilon$
$S \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	

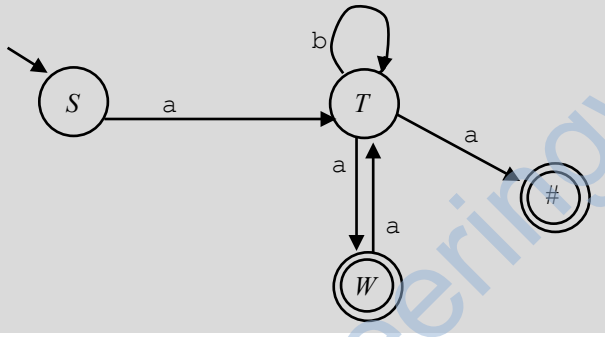
2) Consider the following regular grammar G:

$S \rightarrow aT$
$T \rightarrow bT$
$T \rightarrow a$
$T \rightarrow aW$
$W \rightarrow \varepsilon$
$W \rightarrow aT$

a) Write a regular expression that generates $L(G)$.

$a\ (b \cup aa)\ a$

b) Use *grammartofsm* to generate an FSM M that accepts $L(G)$.



3) Consider again the FSM M shown in Exercise 5.1. Show a regular grammar that generates $L(M)$.

4) Show by construction that, for every FSM M there exists a regular grammar G such that $L(G) = L(M)$.

1. Make M deterministic (to get rid of ε-transitions).
2. Create a nonterminal for each state in the new M.
3. The start state becomes the starting nonterminal
4. For each transition $\delta(T, a) = U$, make a rule of the form $T \rightarrow aU$.
5. For each accepting state T, make a rule of the form $T \rightarrow \varepsilon$.

5) Let $L = \{w \in \{a, b\}^* :$ every a in w is immediately followed by at least one $b\}$.

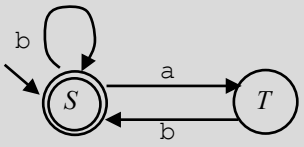
a) Write a regular expression that describes L.

$(ab \cup b)^*$

b) Write a regular grammar that generates L.

$S \rightarrow bS$
$S \rightarrow aT$
$S \rightarrow \varepsilon$
$T \rightarrow bS$

c) Construct an FSM that accepts *L*.



8 Regular and Nonregular Languages

1) For each of the following languages L, state whether or not L is regular. Prove your answer:

a) $\{a^i b^j : i, j \geq 0 \text{ and } i + j = 5\}$.

Regular. A simple FSM with five states just counts the total number of characters.

b) $\{a^i b^j : i, j \geq 0 \text{ and } i - j = 5\}$.

Not regular. L consists of all strings of the form a*b* where the number of a's is five more than the number of b's. We can show that L is not regular by pumping. Let $w = a^{k+5}b^k$. Since $|xy| \leq k$, y must equal a^p for some $p > 0$. We can pump y out once, which will generate the string $a^{k+5-p}b^k$, which is not in L because the number of a's is is less than 5 more than the number of b's.

c) $\{a^i b^j : i, j \geq 0 \text{ and } |i - j| \equiv_5 0\}$.

Regular. Note that $i - j \equiv_5 0$ iff $i \equiv_5 j$. L can be accepted by a straightforward FSM that counts a's (mod 5). Then it counts b's (mod 5) and accepts iff the two counts are equal.

d) $\{w \in \{0, 1, \#\}^* : w = x\#y$, where $x, y \in \{0, 1\}^*$ and $|x| \cdot |y| \equiv_5 0\}$. (Let $\cdot$ mean integer multiplication).

Regular. For $|x| \cdot |y|$ to be divisible by 5, either $|x|$ or $|y|$ (or both) must be divisible by 5. So L is defined by the following regular expression:
$((0 \cup 1)^5)^*\#(0 \cup 1)^* \cup (0 \cup 1)^*\#((0 \cup 1)^5)^*$, where $(0 \cup 1)^5$ is simply a shorthand for writing $(0 \cup 1)$ five times in a row.

e) $\{a^i b^j : 0 \leq i < j < 2000\}$.

Regular. Finite.

f) $\{w \in \{Y, N\}^* : w$ contains at least two Y's and at most two N's$\}$.

Regular. L can be accepted by an FSM that keeps track of the count (up to 2) of Y's and N's.

g) $\{w = xy : x, y \in \{a, b\}^* \text{ and } |x| = |y| \text{ and } \#_a(x) \geq \#_a(y)\}$.

Not regular, which we'll show by pumping. Let $w = a^k bba^k$. y must occur in the first a region and be equal to a^p for some nonzero p. Let $q = 0$. If p is odd, then the resulting string is not in L because all strings in L have even length. If p is even it is at least 2. So both b's are now in the first half of the string. That means that the number of a's in the second half is greater than the number in the first half. So resulting string, $a^{k-p}bba^k$, is not in L.

h) $\{w = xyzy^Rx : x, y, z \in \{a, b\}^*\}$.

Regular. Note that $L = (a \cup b)^*$. Why? Take any string s in $(a \cup b)^*$. Let x and y be ε. Then $s = z$. So the string can be written in the required form. Moral: Don't jump too fast when you see the nonregular "triggers", like ww or ww^R. The entire context matters.

i) $\{w = xyzy : x, y, z \in \{0, 1\}^+\}$.

Regular. Any string w in $\{0, 1\}^+$ is in L iff:

- the last letter of w occurs in at least one other place in the string,
- that place is not the next to the last character,
- nor is it the first character, and
- w contains least 4 letters.

Either the last character is 0 or 1. So:

$$L = ((0 \cup 1)^+ 0 (0 \cup 1)^+ 0) \cup ((0 \cup 1)^+ 1 (0 \cup 1)^+ 1).$$

j) $\{w \in \{0, 1\}^* : \#_0(w) \neq \#_1(w)\}$.

Not regular. This one is quite hard to prove by pumping. Since so many strings are in L, it's hard to show how to pump and get a string that is guaranteed not to be in L. Generally, with problems like this, you want to turn them into problems involving more restrictive languages to which it is easier to apply pumping. So: if L were regular, then the complement of L, L' would also be regular.

$$L' = \{w \in \{0, 1\}^* : \#_0(w) = \#_1(w)\}.$$

It is easy to show, using pumping, that L' is not regular: Let $w = 0^k1^k$. y must occur in the initial string of 0's, since $|xy| \leq k$. So $y = 0^i$ for some $i \geq 1$. Let q of the pumping theorem equal 2 (i.e., we will pump in one extra copy of y). We now have a string that has more 0's than 1's and is thus not in L'. Thus L' is not regular. So neither is L. Another way to prove that L' isn't regular is to observe that, if it were, $L'' = L' \cap 0^*1^*$ would also have to be regular. But L'' is 0^n1^n, which we already know is not regular.

k) $\{w \in \{a, b\}^* : w = w^R\}$.

Not regular, which we show by pumping. Let $w = a^kb^kb^ka^k$. So y must be a^p for some nonzero p. Pump in once. Reading w forward there are more a's before any b's than there are when w is read in reverse. So the resulting string is not in L.

l) $\{w \in \{a, b\}^* : \exists x \in \{a, b\}^+ (w = x\, x^R\, x)\}$.

Not regular, which we show by pumping: Let $w = a^kbba^ka^kb$. y must occur in the initial string of a's, since $|xy| \leq k$. So $y = a^i$ for some $i \geq 1$. Let q of the pumping theorem equal 2 (i.e., we will pump in one extra copy of y). That generates the string $a^{k+i}bba^ka^kb$. If this string is in L, then we must be able to divide it into thirds so that it is of the form $x\, x^R\, x$. Since its total length is $3k + 3 + i$, one third of that (which must be the length of x) is $k + 1 + i/3$. If i is not a multiple of 3, then we cannot carve it up into three equal parts. If i is a multiple of 3, we can carve it up. But then the right boundary of x will shift two characters to the left for every three a's in y. So, if i is just 3, the boundary will shift so that x no longer contains any b's If i is more than 3, the boundary will shift even farther away from the first b. But there are b's in the string. Thus the resulting string cannot be in L. Thus L is not regular.

m) $\{w \in \{a, b\}^*$: the number of occurrences of the substring ab equals the number of occurrences of the substring ba$\}$.

Regular. The idea is that it's never possible for the two counts to be off by more than 1. For example, as soon as there's an ab, there can be nothing but b's without producing the first ba. Then the two counts are equal and will stay equal until the next b. Then they're off by 1 until the next a, when they're equal again. $L = a^*\cup a^+b^+a^+(b^+a^+)^* \cup b^* \cup b^+a^+b^+(a^+b^+)^*$.

n) $\{w \in \{a, b\}^* : w$ contains exactly two more b's than a's$\}$.

Not regular, which we'll show by pumping. Let $w = a^k b^{k+2}$. y must equal a^p for some $p > 0$. Set q to 0 (i.e., pump out once). The number of a's changes, but the number of b's does not. So there are no longer exactly 2 more b's than a's.

o) $\{w \in \{a, b\}^* : w = xyz$, $|x| = |y| = |z|$, and $z = x$ with every a replaced by b and every b replaced by a$\}$. Example: $abbbabbaa \in L$, with $x = abb$, $y = bab$, and $z = baa$.

Not regular, which we'll show by pumping. Let $w = a^k a^k b^k$. This string is in L since $x = a^k$, $y = a^k$, and $z = b^k$. y (from the pumping theorem) $= a^p$ for some nonzero p. Let $q = 2$ (i.e., we pump in once). If p is not divisible by 3, then the resulting string is not in L because it cannot be divided into three equal length segments. If $p = 3i$ for integer i, then, when we divide the resulting string into three segments of equal length, each segment gets longer by i characters. The first segment is still all a's, so the last segment must remain all b's. But it doesn't. It grows by absorbing a's from the second segment. Thus z no longer $= x$ with every a replaced by b and every b replaced by a. So the resulting string is not in L.

p) $\{w$: $w \in \{a - z\}^*$ and the letters of w appear in reverse alphabetical order$\}$. For example, $spoonfeed \in L$.

Regular. L can be recognized by a straightforward 26-state FSM.

q) $\{w$: $w \in \{a - z\}^*$ every letter in w appears at least twice$\}$. For example, $unprosperousness \in L$.

Regular. L can be recognized by an FSM with 26^3 states.

r) $\{w : w$ is the decimal encoding of a natural number in which the digits appear in a non-decreasing order without leading zeros$\}$.

Regular. L can be recognized by an FSM with 10 states that checks that the digits appear in the correct order. Or it can be described by the regular expression: $0^*1^*2^*3^*4^*5^*6^*7^*8^*9^*$.

s) $\{w$ of the form: $<integer_1>+<integer_2>=<integer_3>$, where each of the substrings $<integer_1>$, $<integer_2>$, and $<integer_3>$ is an element of $\{0 - 9\}^*$ and $integer_3$ is the sum of $integer_1$ and $integer_2\}$. For example, $124+5=129 \in L$.

Not regular.

t) L_0^*, where $L_0 = \{ba^i b^j a^k, j \geq 0, 0 \leq i \leq k\}$.

Regular. Both i and j can be 0. So L = $(b^+a^*)^*$.
Regular. Both i and j can be 0. So $L = (b^+a^*)^*$.

u) $\{w : w$ is the encoding (in the scheme we describe next) of a date that occurs in a year that is a prime number$\}$. A date will be encoded as a string of the form $mm/dd/yyyy$, where each m, d, and y is drawn from {0-9}.

Regular. Finite.

v) $\{w \in \{1\}^* : w$ is, for some $n \geq 1$, the unary encoding of $10^n\}$. (So $L = \{1111111111, 1^{100}, 1^{1000}, \ldots\}$.)

Not regular, which we can prove by pumping. Let $w = 1^t$, where t is the smallest integer that is a power of ten and is greater than k. y must be 1^p for some nonzero p. Clearly, p can be at most t. Let $q = 2$ (i.e.,

pump in once). The length of the resulting string s is at most $2t$. But the next power of 10 is $10t$. Thus s cannot be in L.

2) For each of the following languages L, state whether L is regular or not and prove your answer:
   a) $\{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* : \text{in each prefix } x \text{ of } w, \#_{\mathtt{a}}(x) = \#_{\mathtt{b}}(x) = \#_{\mathtt{c}}(x))\}$.

Regular. $L = \{\varepsilon\}$.

   b) $\{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* : \exists \text{ some prefix } x \text{ of } w\ (\#_{\mathtt{a}}(x) = \#_{\mathtt{b}}(x) = \#_{\mathtt{c}}(x))\}$.

Regular. $L = \Sigma^*$, since every string has ε as a prefix.

   c) $\{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* : \exists \text{ some prefix } x \text{ of } w\ (x \neq \varepsilon \text{ and } \#_{\mathtt{a}}(x) = \#_{\mathtt{b}}(x) = \#_{\mathtt{c}}(x))\}$.

Not regular, which we prove by pumping. Let $w = \mathtt{a}^{2k}\mathtt{b}\mathtt{a}^{2k}$.

3) Define the following two languages:
   $L_{\mathtt{a}} = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \text{in each prefix } x \text{ of } w, \#_{\mathtt{a}}(x) \geq \#_{\mathtt{b}}(x)\}$.
   $L_{\mathtt{b}} = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \text{in each prefix } x \text{ of } w, \#_{\mathtt{b}}(x) \geq \#_{\mathtt{a}}(x)\}$.
   a) Let $L_1 = L_{\mathtt{a}} \cap L_{\mathtt{b}}$. Is L_1 regular? Prove your answer.

Regular. $L_1 = \{\varepsilon\}$.

   b) Let $L_2 = L_{\mathtt{a}} \cup L_{\mathtt{b}}$. Is L_2 regular? Prove your answer.

Not regular. First, we observe that $L_2 = \{\varepsilon\} \cup \{w$: the first character of w is an $\mathtt{a}$ and $w \in L_{\mathtt{a}}\}$
$\cup \{w$: the first character of w is a $\mathtt{b}$ and $w \in L_{\mathtt{b}}\}$

We can show that L_2 is not regular by pumping. Let $w = \mathtt{a}^{2k}\mathtt{b}^{2k}$. y must be $\mathtt{a}^p$ for some $0 < p \leq k$. Pump out. The resulting string $w' = \mathtt{a}^{2k-p}\mathtt{b}^{2k}$. Note that w' is a prefix of itself. But it is not in L_2 because it is not ε, nor is it in $L_{\mathtt{a}}$ (because it has more $\mathtt{b}$'s than $\mathtt{a}$'s) or in $L_{\mathtt{b}}$ (because it has $\mathtt{a}$ as a prefix).

4) For each of the following languages L, state whether L is regular or not and prove your answer:
   a) $\{uww^{\mathrm{R}}v : u, v, w \in \{\mathtt{a}, \mathtt{b}\}^+\}$.

Regular. Every string in L has at least 4 characters. Let w have length 1. Then ww^{R} is simply two identical characters next to each other. So L consists of exactly those strings of at least four characters such that there's a repeated character that is not either the first or last. Any such string can be rewritten as u (all the characters up to the first repeated character) w (the first repeated character) w^{R} (the second repeated character) v (all the rest of the characters). So $L = (\mathtt{a} \cup \mathtt{b})^+ (\mathtt{a}\mathtt{a} \cup \mathtt{b}\mathtt{b}) (\mathtt{a} \cup \mathtt{b})^+$.

b) $\{xyzy^R x : x, y, z \in \{a, b\}^+\}$.

Not regular, which we show by pumping. Let $w = a^k babaa^k b$. Note that w is in L because, using the letters from the language definition, $x = a^k b$, $y = a$, and $z = b$. Then y (from the Pumping Theorem) must occur in the first a region. It is a^p for some nonzero p. Set q to 2 (i.e., pump in once). The resulting string is $a^{k+p} babaa^k b$. This string cannot be in L. Since its initial x (from the language definition) region starts with a, there must be a final x region that starts with a. Since the final x region ends with a b, the initial x region must also end with a b. So, thinking about the beginning of the string, the shortest x region is $a^{k+p} b$. But there is no such region at the end of the string unless p is 1. But even in that case, we can't call the final $aa^k b$ string x because that would leave only the middle substring ab to be carved up into yzy^R. But since both y and z must be nonempty, yzy^R must have at least three characters. So the resulting string cannot be carved up into $xyzy^R x$ and so is not in L.

5) Use the Pumping Theorem to complete the proof, given in L.3.1, that English isn't regular.

6) Prove *by construction* that the regular languages are closed under:
   a) intersection.
   b) set difference.

7) Prove that the regular languages are closed under each of the following operations:
   a) $pref(L) = \{w: \exists x \in \Sigma^* (wx \in L)\}$.

By construction. Let $M = (K, \Sigma, \delta, s, A)$ be any FSM that accepts L:
Construct $M' = (K', \Sigma', \delta', s', A')$ to accept $pref(L)$ from M:
1) Initially, let M' be M.
2) Determine the set X of states in M' from which there exists at least one path to some accepting state:
   a) Let n be the number of states in M'.
   b) Initialize X to $\{\}$.
   c) For each state q in M' do:
      For each string w in Σ^* where $0 \le w \le n\text{-}1$ do:
         Starting in state q, read the characters of w one at a time and make the appropriate transition in M'.
         If this process ever lands in an accepting state of M', add q to X and quit processing w.
3) $A_{M'} = X$.

Comments on this algorithm: The only trick here is to find all the states from which there exists some continuation string that could eventually lead to an accepting state. To accept $pref(L)$, that continuation does not have to be present. But we need to know when there could exist one. This can be done by trying continuation strings. But there is an infinite number of them. We can't try them all. True, but we don't have to. If there is a path to an accepting state, then there must be a path that does not require going through a loop. The maximum length of such a path is n-1. So we simply try all strings of length up to n-1.

b) $suff(L) = \{w: \exists x \in \Sigma^* (xw \in L)\}$.

By construction. Let $M = (K, \Sigma, \delta, s, A)$ be any FSM that accepts L. Construct $M' = (K', \Sigma', \delta', s', A')$ to accept $suf(L)$ from M:

1) Initially, let M' be M.
2) Determine the set X of states that are reachable from s:
    a) Let n be the number of states in M'.
    b) Initialize X to $\{s\}$. .
    c) For $i = 1$ to $n - 1$ do:
        i. For every state q in X do:
            a. For every character c in $\Sigma \cup \{\varepsilon\}$ do:
                i. For every transition (q, c, q') in M' do:
                    a. $X = X \cup \{q'\}$
3) Add to M' a new start state s'. Create an ε-transition from s' to s.
4) Add to M' an ε-transition from s' to every element of X.

Comments on this algorithm: The basic idea is that we want to accept the end of any string in L. Another way of thinking of this is that M'' must act as though it had seen the beginning of the string without having actually seen it. So we want to skip from the start state to anyplace that the beginning of any string could have taken us. But we can't necessarily skip to all states of M', since there may be some that aren't reachable from s by any input string. So we must first find the reachable strings.

To find the reachable strings, we must follow all possible paths from s. We need only try strings of length up to n-1 where n is the number of states of M, since any longer strings must simply follow loops and thus cannot get anywhere that some shorter string could not have reached.

We must also be careful that we only skip the beginning of a string. We do not want to be able to skip pieces out of the middle. If the original start state had any transitions into it, we might do that if we allow ε-jumps from it. So we create a new start state s' that functions solely as the start state. We then create the ε-transitions from it to all the states that were marked as reachable.

c) $reverse(L) = \{x \in \Sigma^* : x = w^R \text{ for some } w \in L\}$.

By construction. Let $M = (K, \Sigma, \delta, s, A)$ be any FSM that accepts L. M must be written out completely, without an implied dead state. Then construct $M' = (K', \Sigma', \delta', s', A')$ to accept $reverse(L)$ from M:

1) Initially, let M' be M.
2) Reverse the direction of every transition in M'.
3) Construct a new state q. Make it the start state of M'. Create an ε-transition from q to every state that was an accepting state in M.
4) M' has a single accepting state, the start state of M.

d) letter substitution (as defined in Section 8.3).

Let *sub* be any function from Σ_1 to Σ_2^*. Let *letsub* be a letter substitution function from L_1 to L_2. So *letsub*(L_1) = {$w \in \Sigma_2^*$: $\exists y \in L_1$ and $w = y$ except that every character c of y has been replaced by *sub*(c)}. There are two ways to prove by construction that the regular languages are closed under letter substitution:

First, we can do it with FSMs: if L is regular, then it is accepted by some DFSM $M = (K, \Sigma, \delta, s, A)$. From M we construct a machine $M' = (K', \Sigma', \delta', s', A')$ to accept *letsub*(L). Initially, let $M' = M$. Now change M' as follows. For each arc in M' labelled x, for any x, change the label to *sub*(x). M' will accept exactly the strings accepted by M except that, if M accepted some character x, M' will accept $s(x)$.

Or we can do it with regular expressions: If L is a regular language, then it is generated by some regular expression α. We generate a new regular expression α' that generates *letsub*(L). $\alpha' = \alpha$, except that, for every character c in Σ_1, we replace every occurrence of c in α by *sub*(c).

8) Using the defintions of *maxstring* and *mix* given in Section 8.6, give a precise definition of each of the following languages:

a) *maxstring*(A^nB^n).

maxstring(A^nB^n) = {a^nb^n : $n>0$}. (Note: $\varepsilon \notin$ *maxstring*(A^nB^n) because each element of A^nB^n can be concatenated to ε to generate a string in A^nB^n. But, given any other string in A^nB^n (e.g., aabb), there is nothing except ε that can be added to make a string in A^nB^n.)

b) *maxstring*($a^ib^jc^k$, $1 \le k \le j \le i$).

maxstring($a^ib^jc^k$, $1 \le k \le j \le i$) = {$a^ib^jc^j$, $1 \le j \le i$}.

c) *maxstring*(L_1L_2), where L_1 = {$w \in$ {a, b}* : w contains exactly one a} and L_2 = {a}.

L_1L_2 = b*ab*a. So *maxstring*(L_1L_2) = b*ab*a.

d) *mix*((aba)*).

mix((aba)*) = (abaaba)*.

e) *mix*(a*b*).

This one is tricky. To come up with the answer, consider the following elements of a*b* and ask what elements they generate in *mix*(a*b*):

- aaa: nothing, since only even length strings can contribute.
- aaaa: aaaa. Every even length string of all a's just contributes itself.
- bbb: nothing, since only even length strings can contribute.
- bbbbbb: bbbbbb. Every even length string of all b's just contributes itself.
- aaabbb: aaabbb. If the original string has even length and the number a's is the same as the number of b's, the string contributes itself.
- aab: nothing, since only even length strings can contribute.
- aabbbb: aabbbb. If the original string has even length and the number a's is less than the number of b's, the string contributes itself since the second half is all b's and is thus unchanged by being reversed.
- aaaabb: aaabba. If the original string has even length and the number a's is greater than the number of b's, then the second half starts with a's. Thus reversing the second half creates a substring that starts with b's and ends with a's.

This analysis tells us that *mix*(a*b*) = (aa)* ∪ (bb)* ∪ {a^ib^j, $i \le j$ and $i + j$ is even} ∪ {w : $|w| = n$, n is even, $w = a^ib^ja^k$, $i = n/2$}. But this can be simplified, since the case of all a's is a special case of more a's than b's and the case of all b's is a special case of more b's than a's. So we have:

mix(a*b*) = {a^ib^j, $i \le j$ and $i + j$ is even} ∪ {w : $|w| = n$, n is even, $w = a^ib^ja^k$, $i = n/2$}.

9) Prove that the regular languages are not closed under *mix*.

Let L = (ab)*. Then *mix*((ab)*) = $L' = \{(ab)^{2n+1}, n \ge 0\} \cup \{(ab)^n(ba)^n, n \ge 0\}$. L' is not regular. If it were, then $L'' = L' \cap (ab)^+(ba)^+ = \{(ab)^n(ba)^n, n \ge 1\}$ would also be regular. But it isn't, which we can show by pumping. Let $w = (ab)^N(ba)^N$. y must occur in the $(ab)^N$ region. We consider each possibility:

- $|y|$ is odd. Let $q = 2$. The resulting string has odd length. But all strings in L have even length, so it is not in L.
- $|y|$ is even and $y = (ab)^p$ for some p. Let $q = 2$. The resulting string has more (ab) pairs than (ba) pairs, and so is not in L.
- $|y|$ is even and $y = (ba)^p$ for some p. Let $q = 2$. The resulting string has more (ba) pairs than (ab) pairs, and so is not in L.

10) Recall that *maxstring*(L) = {w: $w \in L$ and $\forall z \in \Sigma^* (z \ne \varepsilon \rightarrow wz \notin L)$}.

a) Prove that the regular languages are closed under *maxstring*.

The proof is by construction. If L is regular, then it is accepted by some DFSA $M = (K, \Sigma, \Delta, s, A)$. We construct a new DFSM $M^* = (K^*, \Sigma^*, \Delta^*, s^*, A^*)$, such that $L(M^*) =$ *maxstring*(L). The idea is that M^* will operate exactly as M would have except that A^* will include only states that are accepting states in M and from which there exists no path of at least one character to any accepting state (back to itself or to any other). So an algorithm to construct M^* is:

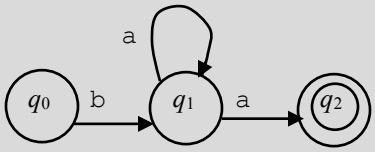
1. Initially, let $M^* = M$.

/* Check each accepting state in M to see whether there are paths from it to some accepting state.

2. For each state q in A do:
   2.1. Follow all paths out of q for $|K|$ steps or until the path reaches an element of A or some state it has already visited.
   2.2. If the path reached an element of A, then q is not an element of A^*.
   2.3. If the path ended without reaching an element of A, then q is an element of A^*.

Comments on this algorithm:

1. Why do we need to start with a deterministic machine? Suppose L is ba*a. *maxstring*(L) = {}. But suppose that M were:



   If we executed our algorithm with this machine, we would accept ba*a rather than {}.
2. Your initial thought may be that the accepting states of M^* should be simply the accepting states of M that have no transitions out of them. But there could be transitions that themselves lead no where, in which case they don't affect the computation of *maxstring*. So we must start by finding exactly those accepting states of M such that there is no continuation (other than ε) that leads again to an accepting state.

b) If *maxstring*(L) is regular, must L also be regular? Prove your answer.

No. Consider $\text{Prime}_a = \{a^n : n \text{ is prime}\}$. Prime_a is not regular. But *maxstring*(Prime_a) = ∅, which is regular.

11) Let *midchar* be a function on languages. Define:
midchar(L) = $\{c : \exists w \in L\ (w = ycz, c \in \Sigma, y \in \Sigma^*, z \in \Sigma^*, |y| = |z|\}$
a) Are the regular languages closed under *midchar*? Prove your answer.

Yes. For any language L, *midchar*(L) must be finite and thus regular.

b) Are the nonregular languages closed under *midchar*? Prove your answer.

No. A^nB^n is not regular, but *midchar*(A^nB^n) = ∅, which is regular.

12) Define the function *twice*(L) = $\{w : \exists x \in L$ (x can be written as $c_1c_2 \ldots c_n$, for some $n \geq 1$, where each $c_i \in \Sigma_L$, and $w = c_1c_1c_2c_2 \ldots c_nc_n)\}$.
a) Let $L = (1 \cup 0)^*1$. Write a regular expression for *twice*(L).

$(11 \cup 00)^*11$.

b) Are the regular languages closed under *twice*? Prove your answer.

Yes, by construction. If L is regular, then there is some DFSM M that accepts it. We build an FSM M' that accepts *twice*(L):

1. Initially, let M' be M.
2. Modify M' as follows: For every transition in M from some state p to some state q with label c, do:
   2.1. Remove the transition from M'.
   2.2. Create a new state p'.
   2.3. Add to M' two transitions: $((p, c), p')$ and $((p', c), q)$.
3. Make the start state of M' be the same as the start state of M.
4. Make every accepting state in M also be an accepting state in M'.

13) Define the function *shuffle*(L) = $\{w : \exists x \in L$ (w is some permutation of x)$\}$. For example, if $L = \{ab, abc\}$, then *shuffle*(L) = {ab, abc, ba, acb, bac, bca, cab, cba}. Are the regular languages closed under *shuffle*? Prove your answer.

No. Let $L = (ab)^*$. Then *shuffle*(L) = $\{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$, which is not regular.

14) Define the function *copyandreverse*(L) = $\{w : \exists x \in L\ (w = xx^R)\}$. Are the regular languages closed under *copyandreverse*? Prove your answer.

No. Let $L = (a \cup b)^*$. Then *copyandreverse*(L) = WW^R, which is not regular.

15) Let L_1 and L_2 be regular languages. Let L be the language consisting of strings that are contained in exactly one of L_1 and L_2. Prove that L is regular.

We prove this by construction of an FSM that accepts L. Let M_1 be a DFSM that accepts L_1 and let M_2 be a DFSM that accepts L_2. The idea is that we'll build a machine M' that simulates the parallel execution of M_1 and M_2. The states of M' will correspond to ordered pairs of the states of M_1 and M_2. The accepting states of M' will be the ones that correspond to ordered pairs of states that are accepting in their original machines.

16) Define two integers i and j to be ***twin primes*** 💻 iff both i and j are prime and $|j - i| = 2$.

a) Let $L = \{w \in \{1\}^* : w$ is the unary notation for a natural number n such that there exists a pair p and q of twin primes, both $> n.\}$ Is L regular?

Regular. We don't know how to build an FSM for L. 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 (since it contains exactly the unary encodings of those values of n). Since L is finite, it is regular.

It is unknown which of these is true.

b) Let $L = \{x, y : x$ is the decimal encoding of a positive integer i, y is the decimal encoding of a positive integer j, and i and j are twin primes$\}$. Is L regular?

It is unknown whether the number of pairs of twin primes is finite. If it is, then L is regular. If it is not, then L is probably not regular.

17) Consider any function $f(L_1) = L_2$, where L_1 and L_2 are both languages over the alphabet $\Sigma = \{0,1\}$. We say that function f is ***nice*** iff L_2 is regular iff L_1 is regular. For each of the following functions, state whether or not it is nice and prove your answer:

a) $f(L) = L^R$.

Nice. The regular languages are closed under reverse.

b) $f(L) = \{w: w$ is formed by taking a string in L and replacing all 1's with 0's and leaving the 0's unchanged$\}$.

Not nice. Let L be $\{0^n1^n: n \geq 0\}$, which is not regular. Then $f(L) = (00)^*$, which is regular.

c) $f(L) = L \cup 0^*$

Not nice. Let L be $\{0^p: p$ is prime$\}$, which is not regular. Then $f(L) = 0^*$, which is regular.

d) $f(L) = \{w: w$ is formed by taking a string in L and replacing all 1's with 0's and all 0's with 1's (simultaneously)$\}$.

Nice. The regular languages are closed under letter substitution.

e) $f(L) = \{w: \exists x \in L\ (w = x00)\}$.

Nice. If L is regular, then $f(L)$ is the concatenation of two regular languages and so is regular. If $f(L)$ is regular, then there is some FSM M that accepts it. From M, we construct a new FSM M' that accepts L', defined to be the set of strings with the property that if 00 is concatenated onto them they are in $f(L)$. To build M', we begin by letting it be M. Then we start at each accepting state of M and trace backwards along arcs labeled 0. The set of states that can be reached by following exactly two steps in that way become the accepting states of M'. Since there is an FSM that accepts L', it is regular. Finally, we note that $L' = L$.

f) $f(L) = \{w: w$ is formed by taking a string in L and removing the last character$\}$.

Not nice. Let L be $\{a^pb: p$ is prime$\} \cup \{a^pc: p$ is a positive integer and is not prime$\}$, which is not regular. Then $f(L) = a^+$, which is regular.

18) We'll say that a language L over an alphabet Σ is ***splitable*** iff the following property holds: Let w be any string in L that can be written as $c_1c_2 \ldots c_{2n}$, for some $n \geq 1$, where each $c_i \in \Sigma$. Then $x = c_1c_3 \ldots c_{2n-1}$ is also in L.

a) Give an example of a splitable regular language.

Two simple examples are $(\mathtt{a} \cup \mathtt{b})^*$ and $\mathtt{a}^*$.

b) Is every regular language splitable?

No. $L = (\mathtt{ab})^*$ is regular. Let $w = \mathtt{abab}$. Then $w \in L$. The split version of w is $\mathtt{aa}$, which is not in L. So L isn't splitable. An even simpler example is $L = \{\mathtt{ab}\}$.

c) Does there exist a nonregular language that is splitable?

Yes. One example is $L = A^nB^nC^n = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$. Only even length strings in L have an impact on whether or not L is splitable. So consider any string w in $\{\mathtt{a}^{2n}\mathtt{b}^{2n}\mathtt{c}^{2n} : n \geq 0\}$. The split version of any such string is $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, which is also in L.

Another example is $L = \{\mathtt{a}^n : n > 2$ and n is prime$\}$. Since L contains no even length strings, it is trivially splitable.

19) Define the class IR to be the class of languages that are both infinite and regular. Is the class IR closed under:

a) union.

Yes. Let $L_3 = L_1 \cup L_2$. If L_1 and L_2 are in IR, then L_3 must be. First we observe that L_3 must be infinite because every element of L_1 is in L_3. Next we observe that L_3 must be regular because the regular languages are closed under union.

b) intersection.

No. Let $L_1 = \mathtt{a}^*$. Let $L_2 = \mathtt{b}^*$. Both L_1 and L_2 are infinite and regular. But $L_3 = L_1 \cap L_2 = \{\varepsilon\}$, which is finite.

c) Kleene star.

Yes. Let $L_2 = L_1^*$. If L_1 is in IR, then L_2 must be. First we observe that L_2 must be infinite because every element of L_1 is in L_2. Next we observe that L_2 must be regular because the regular languages are closed under Kleene star.

20) Consider the language $L = \{x\mathtt{0}^n y\mathtt{1}^n z : n \geq 0, x \in P, y \in Q, z \in R\}$, where P, Q, and R are nonempty sets over the alphabet $\{\mathtt{0}, \mathtt{1}\}$. Can you find regular languages P, Q, and R such that L is not regular? Can you find regular languages P, Q, and R such that L is regular?

Let P, Q, $R = \{\varepsilon\}$. Then $L = \mathtt{0}^n\mathtt{1}^n$, and so is not regular. On the other hand, let $P = \mathtt{0}^*$, $Q = \{\varepsilon\}$ and let $R = \mathtt{1}^*$. Now $L = \mathtt{0}^*\mathtt{1}^*$, which is regular.

21) For each of the following claims, state whether it is *True* or *False*. Prove your answer.:

a) There are uncountably many non-regular languages over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

True. There are uncountably many languages over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Only a countably infinite number of those are regular.

b) The union of an infinite number of regular languages must be regular.

False. Let $L = \cup$ ({ε}, {ab}, {aabb}, {aaabbb}, …) Each of these languages is finite and thus regular. But the infinite union of them is $\{a^n b^n , n \geq 0\}$, which is not regular.

c) The union of an infinite number of regular languages is never regular.

Nothing says the languages that are being unioned have to be different. So, Let $L = \cup$ (a*, a*, a*, …), which is a*, which is regular.

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

False. Let $L_1 = \{a^p : p$ is prime$\}$. Let $L_2 = \{a^p : p$ is greater than 0 and not prime$\}$. Neither L_1 nor L_2 is regular. But $L_1 \cup L_2 = a^+$, which is regular.

e) If L_1 and L_2 are regular languages, then $L_1 \otimes L_2 = \{w : w \in (L_1 - L_2)$ or $w \in (L_2 - L_1)\}$ is regular.

True. The regular languages are closed under union and set difference.

f) If L_1 and L_2 are regular languages and $L_1 \subseteq L \subseteq L_2$, then L must be regular.

False. Let $L_1 = \varnothing$. Let $L_2 = (a \cup b)^*$. Let $L = \{a^n b^n : n \geq 0\}$, which is not regular.

g) The intersection of a regular language and a nonregular language must be regular.

False. Let $L_1 = (a \cup b)^*$, which is regular. Let $L_2 = \{a^n b^n , n \geq 0\}$, which is not regular. Then $L_1 \cap L_2 = \{a^n b^n , n \geq 0\}$, which is not regular. A simpler example is: Let $L_1 = a^*$, which is regular. Let $L_2 = \{a^p : p$ is prime$\}$, which is not regular. $L_1 \cap L_2 = L_2$.

h) The intersection of a regular language and a nonregular language must not be regular.

False. Let $L_1 = \{ab\}$ (regular). Let $L_2 = \{a^n b^n , n \geq 0\}$ (not regular). $L_1 \cap L_2 = \{ab\}$, which is regular.

i) The intersection of two nonregular languages must not be regular.

False. Let $L_1 = \{a^p : p$ is prime$\}$, which is not regular.. Let $L_2 = \{b^p : p$ is prime$\}$, which is also not regular.. $L_1 \cap L_2 = \varnothing$, which is regular.

j) The intersection of a finite number of nonregular languages must not be regular.

False. Since two is a finite number, we can used the same counterexample that we used above in part i. Let $L_1 = \{a^p : p$ is prime$\}$, which is not regular.. Let $L_2 = \{b^p : p$ is prime$\}$, which is also not regular.. $L_1 \cap L_2 = \varnothing$, which is regular.

k) The intersection of an infinite number of regular languages must be regular.

False. Let $x_1, x_2, x_3, \ldots$ be the sequence 0, 1, 4, 6, 8, 9, … of nonprime, nonnegative integers. Let a^{xi} be a string of x_i a's. Let L_i be the language $a^* - \{a^{xi}\}$.

Now consider L = the infinite intersection of the sequence of languages $L_1, L_2, \ldots$ Note that $L = \{a^p$, where p is prime$\}$. We have proved that L is not regular.

l) It is possible that the concatenation of two nonregular languages is regular.

True. To prove this, we need one example: Let $L_1 = \{a^i b^j, i, j \geq 0$ and $i \neq j\}$. Let $L_2 = \{b^n a^m, n, m \geq 0$ and $n \neq m$. $L_1L_2 = \{a^i b^j a^k$ such that:

- If i and k are both 0 then $j \geq 2$.
- If $i = 0$ and $k \neq 0$ then $j \geq 1$.
- If $i \neq 0$ and $k = 0$ then $j \geq 1$.
- If i and k are both 1 then $j \neq 1$.
- Otherwise, $j \geq 0\}$.

In other words, L_1L_2 is almost a*b*a*, with a small number of exceptions that can be checked for by a finite state machine.

m) It is possible that the union of a regular language and a nonregular language is regular.

True. Let $L_1 = a^*$, which is regular. Let $L_2 = \{a^p$: $2 \leq p$ and p is prime$\}$, which is not regular. $L_1 \cup L_2 = a^*$, which is regular.

n) Every nonregular language can be described as the intersection of an infinite number of regular languages.

True. Note first that every nonregular language is countably infinite. (Because every finite language is regular and no language contains more strings than Σ*.)

Let L be a non-regular language, and consider the following infinite set of languages:

$$S = \{L_i \mid L_i \text{ is the language } \Sigma^* - w_i\}$$

where w_i is the i-th string in Σ* in lexicographical order. For example, suppose Σ= {a,b} and the words are ordered as follows: ε, a, b, aa, ab, ba, bb, aaa, ...Then, L_1=Σ*-{ε}, L_2 = Σ*-{a}, L_3=Σ*-{b}, L_4=Σ*-{aa}, etc. Obviously, each L_i is a regular language (since it's the complement of the regular language that contains only one string, namely the i-th string m, which is regular).

Now, consider the following set of languages: $T = \{L_i \mid$ the i-th string in Σ* is not in $L\}$, i.e., T is the set of all L_i's where the i-th string (the one that defines L_i) is not in L. For example, suppose $L = \{a^n b^n : n \geq 0\}$. Since Σ = {a,b}, the L_i's are as mentioned above. T = {Σ*- a, Σ*- b, Σ*- aa, Σ*- ba, Σ*- bb, Σ*- aaa, …}.

Return now to our original, nonregular language L. We observe that it is simply the intersection of all of the languages in T. What we are effectively doing is subtracting out, one at a time, all the strings that should be excluded from L.

Note that T is infinite (i.e., it contains an infinite number of languages). Why? Recall that L = Σ* - one string for each element of T. If T were finite, that would mean that L were equal to Σ* - L', where L' contains only some finite number of strings. But then L' would be regular, since it would be finite. If L' were regular, then its complement, Σ* - L', would also be regular. But L is precisely the complement of L' and, by hypothesis, L is not regular.

This result may seem a bit counterintuitive. After all, we have proven that the regular languages are closed under intersection. But what we proved is that the regular languages are closed under *finite* intersection. It is true that if you intersect any finite number of regular languages, the result must be regular. We can prove that by induction on the number of times we apply the intersection operator, starting with the base case, that we proved by construction, in which we intersect any two regular languages and must get a result that is regular. But the induction proof technique only applies to finite values of n. To see this, consider a more straightforward case. We can prove, by induction, that the sum of the integers from 1 to N is $N*(N+1)/2$. So, for any value of N, we can determine the sum. But what is the sum of all the integers? In

other words, what is the answer if we sum an infinite number of integers. The answer is that it is not an integer. There exists no value N such that the answer is $N*(N+1)/2$.

o) If L is a language that is not regular, then L^* is not regular.

False.
Let $L = \text{Prime}_a = \{a^n : n \text{ is prime}\}$. L is not regular.
$L^* = \{\varepsilon\} \cup \{a^n : 1 < n\}$. L^* is regular.

p) If L^* is regular, then L is regular.

False.
Let $L = \text{Prime}_a = \{a^n : n \text{ is prime}\}$. L is not regular.
$L^* = \{\varepsilon\} \cup \{a^n : 1 < n\}$. L^* is regular.

q) The nonregular languages are closed under intersection.

False.
Let $L_1 = \{a^n b^n$, where n is even$\}$. L_1 is not regular.
Let $L_2 = \{a^n b^n$, where n is odd$\}$. L_2 is not regular.
$L = L_1 \cap L_2 = \varnothing$, which is regular.

r) Every subset of a regular language is regular.

False.
Let $L = a^*$, which is regular.
Let $L' = a^p$, where p is prime. L' is not regular, but it is a subset of L.

s) Let $L_4 = L_1L_2L_3$. If L_1 and L_2 are regular and L_3 is not regular, it is possible that L_4 is regular.

True. Example:

Let $L_1 = \{\varepsilon\}$.	L_1 is regular.
Let $L_2 = a^*$.	L_2 is regular.
Let $L_3 = a^p$, where p is prime.	L_3 is not regular.
$L_4 = a^k$, where $k \geq 2$	L_4 is regular, because it is defined by aaa*.

t) If L is regular, then so is $\{xy : x \in L \text{ and } y \notin L\}$.

True. $\{xy : x \in L \text{ and } y \notin L\}$ can also be described as the concatenation of L and $\neg L$. Since the regular languages are closed under complement, this means that it is the concatenation of two regular languages. The regular languages are closed under complement.

u) Every infinite regular language properly contains another infinite regular language.

True. Let L be an infinite regular language and let w be a string in L. Then $L - \{w\}$ is also both infinite and regular.

9 Decision Procedures for Regular Languages

1) Define a decision procedure for each of the following questions. Argue that each of your decision procedures gives the correct answer and terminates.

a) Given two DFSMs M_1 and M_2, is $L(M_1) = L(M_2)^R$?

1. From M_1, construct M_R, which accepts $L(M_1)^R$, using the algorithm presented in the solution to Exercise 8.7 (c).
2. Determine whether M_1 and M_2 accept the same language, using the algorithm *equalFSMs*, presented in Section 9.1.4.

b) Given two DFSMs M_1 and M_2 is $|L(M_1)| < |L(M_2)|$?

1. If $L(M_1)$ is infinite, return *False*.
2. If $L(M_2)$ is infinite, return *True*.

/* Now we know that both languages are finite. We need to count the number of elements in each.

3. Run every string in Σ_{M1}* of length less than $|K_{M1}|$ through M_1, counting the number that are accepted. Call that number C_1.
4. Run every string in Σ_{M2}* of length less than $|K_{M2}|$ through M_2, counting the number that are accepted. Call that number C_2.
5. If $C_1 < C_2$ then return *True*. Else return *False*.

c) Given a regular grammar G and a regular expression α, is $L(G) = L(\alpha)$?

1. From G, create a NDFSM M_G that accepts $L(G)$.
2. From α, create an FSM M_α that accepts $L(\alpha)$.
3. Determine whether M_G and M_α are equivalent.

d) Given two regular expressions, α and β, do there exist any even length strings that are in $L(\alpha)$ but not $L(\beta)$?

e) Let $\Sigma = \{a, b\}$ and let α be a regular expression. Does the language generated by α contains all the even length strings in Σ*.

f) Given an FSM M and a regular expression α, is it true that $L(M)$ and $L(\alpha)$ are both finite and M accepts exactly two more strings than α generates?

1. From α, build FSM P, such that $L(P) = L(\alpha)$.
2. If $L(M)$ is infinite, return *False*.
3. If $L(P)$ is infinite, return *False*.

/* Now we know that both languages are finite. Thus neither machine accepts any strings of length equal to or greater than the number of states it contains. So we simply count the number of such "short" strings that each machine accepts.

4. Run every string in Σ_M* of length less than $|K_M|$ through M, counting the number that are accepted. Call that number C_M.
5. Run every string in Σ_P* of length less than $|K_P|$ through P, counting the number that are accepted. Call that number C_P.
6. If $C_M = C_P+2$, return *True*; else return *False*.

g) Let Σ = {a, b} and let α and β be regular expressions. Is the following sentence true:

$$(L(\beta) = \mathtt{a}^*) \vee (\forall w\,(w \in \{\mathtt{a}, \mathtt{b}\}^* \wedge |w| \text{ even}) \rightarrow w \in L(\alpha))$$

1. From β, build FSM M_1. Make it deterministic. Minimize it, producing M_2.
2. Build M_a, the simple one-state machine that accepts a*.
3. If M_2 and M_a are identical except for state names then return true. Else continue.

/* Observe that the second condition says that L_E, the language of even length strings of a's and b's, is a subset of $L(\alpha)$. This is equivalent to saying that L_E - $L(\alpha)$ is empty. So:
4. From α, build FSM M_3.
5. Build M_E that accepts exactly L_E, the language of even length strings of a's and b's.
6. Build M_D that accepts $L(M_E)$ - $L(M_3)$.
7. See if $L(M_D)$ is empty. If it is, return true. Else return false.

h) Given a regular grammar G, is $L(G)$ regular?

i) Given a regular grammar G, does$G$ generate any odd length strings?

10 Summary and References

Part III: Context-Free Languages and Pushdown Automata

11 Context-Free Grammars

1) Let Σ = {a, b}. For the languages that are defined by each of the following grammars, do each of the following:
   i. List five strings that are in *L*.
   ii. List five strings that are not in *L*.
   iii. Describe *L* concisely. You can use regular expressions, expressions using variables (e.g., $a^n b^n$, or set theoretic expressions (e.g., {*x*: …})
   iv. Indicate whether or not *L* is regular. Prove your answer.

   a) $S \rightarrow aS \mid Sb \mid \varepsilon$

   i. ε, a, b, aaabbbb, ab
   ii. ba, bbaa, bbbbba, ababab, aba
   iii. *L* = a*b*.
   iv. *L* is regular because we can write a regular expression for it.

   b) $S \rightarrow aSa \mid bSb \mid a \mid b$

   i. a, b, aaa, bbabb, aaaabaaaa
   ii. ε, ab, bbbbbbba, bb, bbbaaa
   iii. *L* is the set of odd length palindromes, i.e., $L = \{w = x\,(a \cup b)\,x^R$, where $x \in \{a,b\}^*\}$.
   iv. *L* is not regular. Easy to prove with pumping. Let $w = a^k baba^k$. *y* must be in the initial a region. Pump in and there will no longer be a palindrome.

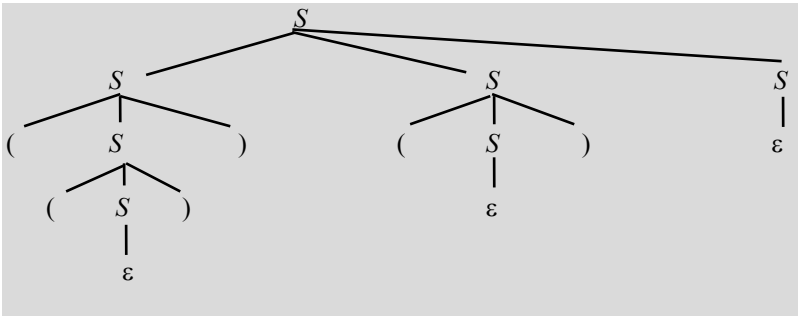
   c) $S \rightarrow aS \mid bS \mid \varepsilon$

   i. ε, a, aa, aaa, ba
   ii. There aren't any over the alphabet {a, b}.
   iii. *L* = (a ∪ b)*.
   iv. *L* is regular because we can write a regular expression for it.

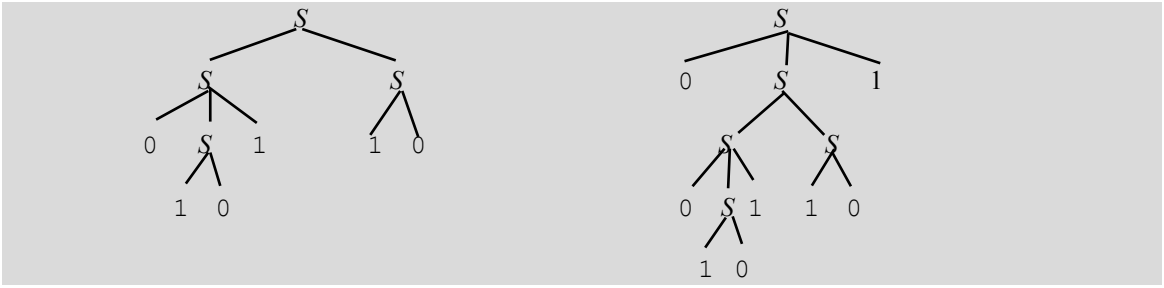
   d) $S \rightarrow aS \mid aSbS \mid \varepsilon$

   i. ε, a, aaa, aaba, aaaabbbb
   ii. b, bbaa, abba, bb
   iii. $L = \{w \in \{a, b\}^* : \text{in all prefixes } p \text{ of } w, \#_a(p) \geq \#_b(p)\}$.
   iv. *L* isn't regular. Easy to prove with pumping. Let $w = a^k b^k$. *y* is in the a region. Pump out and there will be fewer a's than b's.

2) Let *G* be the grammar of Example 11.12. Show a third parse tree that *G* can produce for the string (())().



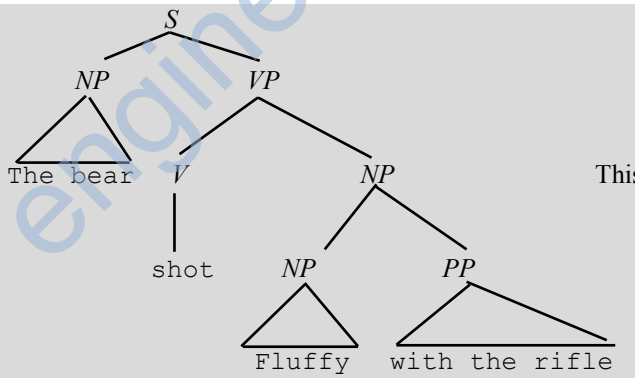
3) Consider the following grammar G: $S \rightarrow 0S1 \mid SS \mid 10$
Show a parse tree produced by G for each of the following strings:
a) 010110
b) 00101101



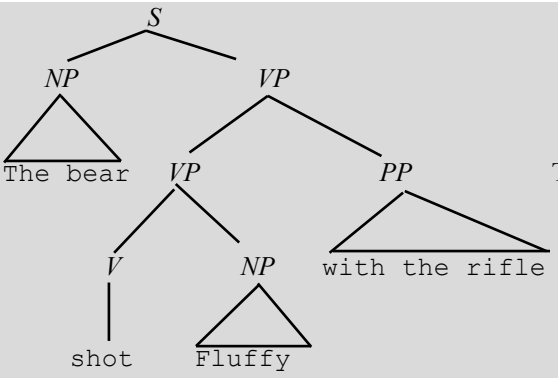
4) Consider the following context free grammar G:
$S \rightarrow aSa$
$S \rightarrow T$
$S \rightarrow \varepsilon$
$T \rightarrow bT$
$T \rightarrow cT$
$T \rightarrow \varepsilon$
One of these rules is redundant and could be removed without altering $L(G)$. Which one?

$S \rightarrow \varepsilon$

5) Using the simple English grammar that we showed in Example 11.6, show two parse trees for each of the following sentences. In each case, indicate which parse tree almost certainly corresponds to the intended meaning of the sentence:
a) The bear shot Fluffy with the rifle.

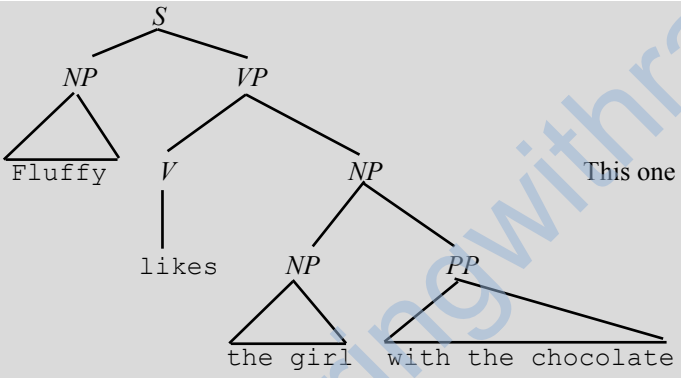


This one doesn't make much sense.

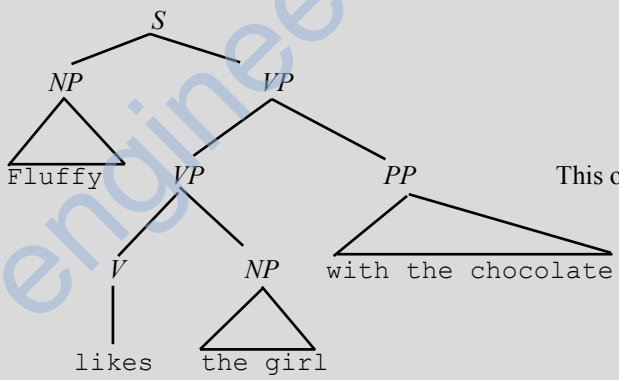


This one makes sense; the rifle was used for shooting.

b) `Fluffy likes the girl with the chocolate.`



This one makes sense.



This one doesn't make sense.

6) Show a context-free grammar for each of the following languages L:

a) BalDelim = $\{w$: where w is a string of delimeters: (,), [,], {, }, that are properly balanced$\}$.

$S \rightarrow (S) \mid [S] \mid \{ S \} \mid S S \mid \varepsilon$

b) $\{a^i b^j : 2i = 3j + 1\}$.

$S \rightarrow aaaSbb \mid aab$

c) $\{a^i b^j : 2i \neq 3j + 1\}$.

We can begin by analyzing L, as shown in the following table:

# of a's	Allowed # of b's
0	any
1	any
2	any except 1
3	any
4	any
5	any except 3
6	any
7	any
8	any except 5

$S \rightarrow$ aaa$S$bb
$S \rightarrow$ aaa$X$ /* extra a's
$S \rightarrow T$ /* terminate
$X \rightarrow A \mid A$ b /* arbitrarily more a's
$T \rightarrow A \mid B \mid$ a $B \mid$ aabb B /* note that if we add two more a's we cannot add just a single b.
$A \rightarrow$ a $A \mid \varepsilon$
$B \rightarrow$ b $B \mid \varepsilon$

d) $\{w \in \{a, b\}^* : \#_a(w) = 2\ \#_b(w)\}$.

$S \rightarrow S$aSaSbS
$S \rightarrow S$aSbSaS
$S \rightarrow S$bSaSaS
$S \rightarrow \varepsilon$

e) $\{w \in \{a, b\}^* : w = w^R\}$.

$S \rightarrow$ aSa
$S \rightarrow$ bSb
$S \rightarrow \varepsilon$
$S \rightarrow$ a
$S \rightarrow$ b }

f) $\{a^i b^j c^k : i, j, k \geq 0$ and $(i \neq j$ or $j \neq k)\}$.

$S \rightarrow XC$ /* $i \neq j$
$S \rightarrow AY$ /* $j \neq k$
$X \rightarrow$ aXb
$X \rightarrow A'$ /* at least one extra a
$X \rightarrow B'$ /* at least one extra b
$Y \rightarrow$ bYc $\mid B' \mid C'$
$A' \rightarrow$ a $A' \mid$ a
$B' \rightarrow$ b $B' \mid$ b
$C' \rightarrow$ c $C' \mid$ c
$A \rightarrow$ a $A \mid \varepsilon$
$C \rightarrow$ c $C \mid \varepsilon$ }

g) $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (k \leq i \text{ or } k \leq j)\}$.

$S \rightarrow A \mid B$
$A \rightarrow aAc \mid aA \mid M$
$B \rightarrow aB \mid F$
$F \rightarrow bFc \mid bF \mid \varepsilon$
$M \rightarrow bM \mid \varepsilon$

h) $\{w \in \{a, b\}^* : \text{every prefix of } w \text{ has at least as many a's as b's}\}$.

$S \rightarrow aS \mid aSb \mid SS \mid \varepsilon$

i) $\{a^n b^m : m \geq n, m\text{-}n \text{ is even}\}$.

$S \rightarrow aSb \mid S \rightarrow Sbb \mid \varepsilon$

j) $\{a^m b^n c^p d^q : m, n, p, q \geq 0 \text{ and } m + n = p + q\}$.

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 is 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:

$S \rightarrow aSd$
$S \rightarrow T$
$S \rightarrow U$
$T \rightarrow aTc$
$T \rightarrow V$
$U \rightarrow bUd$
$U \rightarrow V$
$V \rightarrow bVc$
$V \rightarrow \varepsilon$

k) $\{xc^n : x \in \{a, b\}^* \text{ and } (\#_a(x) = n \text{ or } \#_b(x) = n)\}$.

$S \rightarrow A \mid B$
$A \rightarrow B' a B' A c \mid B'$
$B' \rightarrow bB' \mid \varepsilon$
$B \rightarrow A' b A' B c \mid A'$
$A' \rightarrow aA' \mid \varepsilon$

l) $\{b_i \# b_{i+1}^R : b_i$ is the binary representation of some integer i, $i \geq 0$, without leading zeros$\}$. (For example $101\#011 \in L$.)

L can be written as:

$$0\#1 \cup \{1^k\#0^k1 : k > 0\} \cup \{u01^k\#0^k1u^R : k \geq 0 \text{ and } u \in 1(0 \cup 1)^*\}$$

So a grammar for L is:

$$S \rightarrow 0\#1 \mid 1\ S_1\ 1 \mid 1\ S_2\ 1$$
$$S_1 \rightarrow 1\ S_1\ 0 \mid \#0$$
$$S_2 \rightarrow 1\ S_2\ 1 \mid 0\ S_2\ 0 \mid 0\ A\ 1$$
$$A \rightarrow 1\ A\ 0 \mid \#$$

m) $\{x^R\#y : x, y \in \{0, 1\}^*$ and x is a substring of $y\}$.

$$S \rightarrow S0 \mid S1 \mid S_1$$
$$S_1 \rightarrow 0S_10 \mid 1S_11 \mid T$$
$$T \rightarrow T1 \mid T0 \mid \#$$

7) Let G be the ambiguous expression grammar of Example 11.14. Show at least three different parse trees that can be generated from G for the string id+id*id*id.

8) Consider the unambiguous expression grammar G' of Example 11.19.

a) Trace a derivation of the string id+id*id*id in G'.

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow \mathtt{id} + T \Rightarrow \mathtt{id} + T * F \Rightarrow \mathtt{id} + T * F * F \Rightarrow \mathtt{id} + F * F * F \Rightarrow$$
$$\mathtt{id} + \mathtt{id} * F * F \Rightarrow \mathtt{id} + \mathtt{id} * \mathtt{id} * F \Rightarrow \mathtt{id} + \mathtt{id} * \mathtt{id} * \mathtt{id}$$

b) Add exponentiation (**) and unary minus (-) to G', assigning the highest precedence to unary minus, followed by exponentiation, multiplication, and addition, in that order.

$$R = \{ E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow F ** X$$
$$F \rightarrow X$$
$$X \rightarrow -X$$
$$X \rightarrow Y$$
$$Y \rightarrow (E)$$
$$Y \rightarrow \mathtt{id} \}.$$

9) Let $L = \{w \in \{a, b, \cup, \varepsilon, (,), *, ^+\}^* : w$ is a syntactically legal regular expression$\}$.

a) Write an unambiguous context-free grammar that generates L. Your grammar should have a structure similar to the arithmetic expression grammar G' that we presented in Example 11.19. It should create parse trees that:

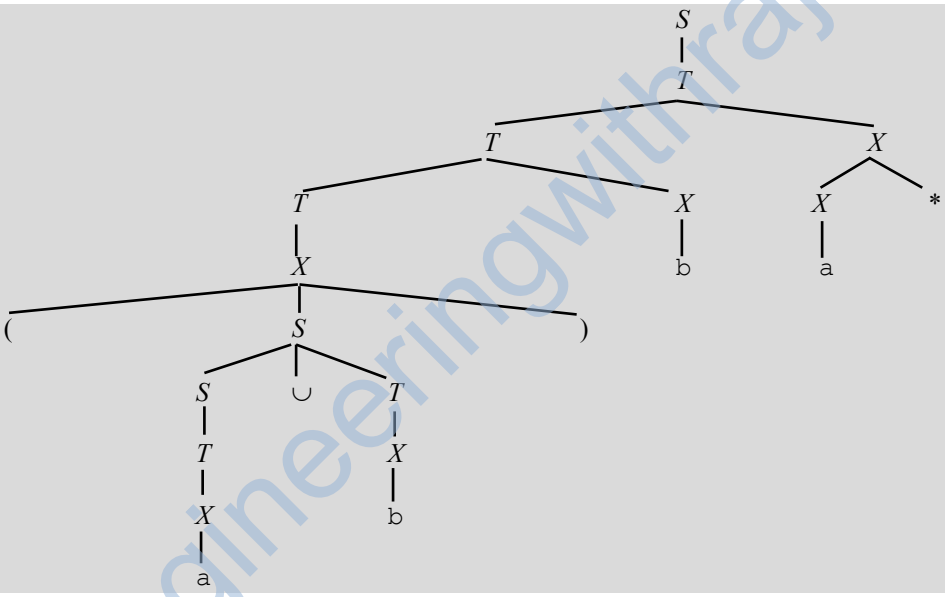
- Associate left given operators of equal precedence, and
- Correspond to assigning the following precedence levels to the operators (from highest to lowest):
  - * and $^+$
  - concatenation
  - $\cup$

One problem here is that we want the symbol ε to be in Σ. But it is also generally a metasymbol in our rule-writing language. If we needed to say that a rule generates the empty string, we could use "" instead

of ε. As it turns out, in this grammar we won't need to do that. We will, in this grammar, treat the symbol ε as a terminal symbol, just like ∪.

$S \rightarrow S \cup T$
$S \rightarrow T$
$T \rightarrow T\,X$
$T \rightarrow X$
$X \rightarrow X$*
$X \rightarrow X^{+}$
$X \rightarrow$ a
$X \rightarrow$ b
$X \rightarrow \varepsilon$
$X \rightarrow (S)$

b) Show the parse tree that your grammar will produce for the string (a ∪ b) ba*.



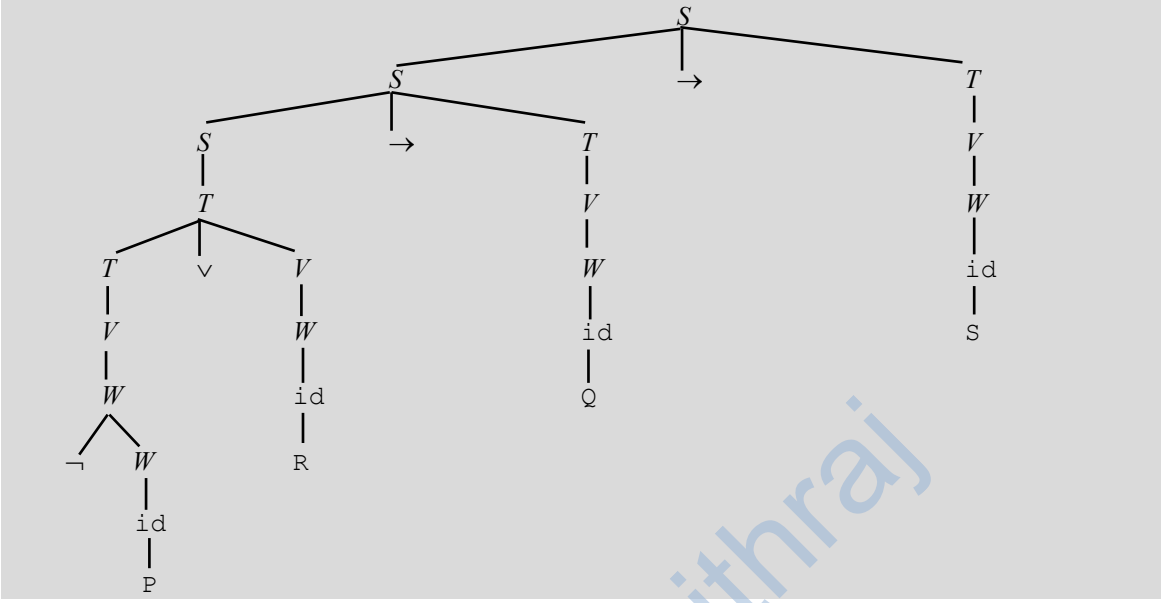
10) Let $L = \{w \in \{$A - Z, ¬, ∧, ∨, →, (,)$\}^* : w$ is a syntactically legal Boolean formula$\}$.

a) Write an unambiguous context-free grammar that generates L and that creates parse trees that:

- Associate left given operators of equal precedence, and
- Correspond to assigning the following precedence levels to the operators (from highest to lowest): ¬, ∧, ∨, and →.

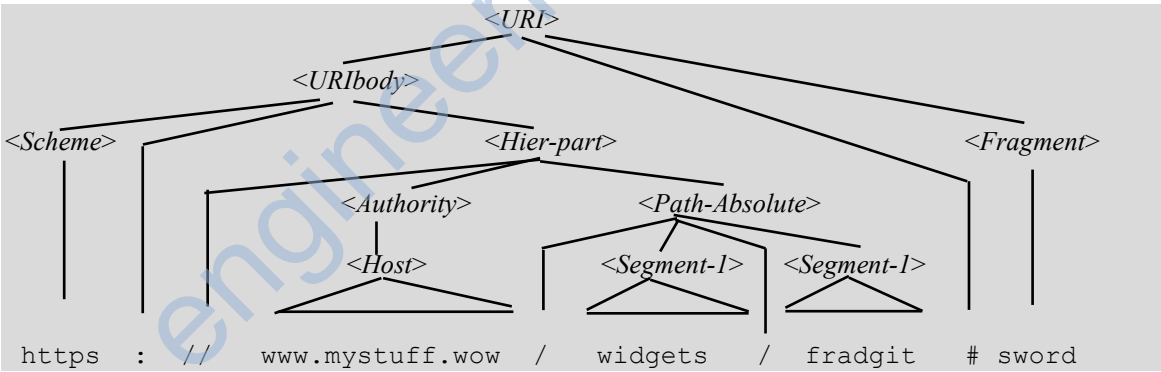
$S \rightarrow S \rightarrow T \mid T$
$T \rightarrow T \vee V \mid V$
$V \rightarrow V \wedge W \mid W$
$W \rightarrow \neg W \mid$ id $\mid (S)$

b) Show the parse tree that your grammar will produce for the string ¬P ∨ R → Q → S.



11) In I.3.1, we present a simplified grammar for URIs (Uniform Resource Identifiers), the names that we use to refer to objects on the Web.
a) Using that grammar, show a parse tree for:

`https://www.mystuff.wow/widgets/fradgit#sword`



b) Write a regular expression that is equivalent to the grammar that we present.

We can build it by recursively expanding the nonterminals of the grammar. We can do this because there are no recursive rules. We'll simply leave the nonterminals that aren't expanded in detail in the grammar. So we get:

<*URIbody*> ($\varepsilon \cup \#$ <*Fragment*>)

<*Scheme*> : <*Hier-part*> ($\varepsilon \cup$ (? <*Query*>)) ($\varepsilon \cup \#$ <*Fragment*>)

(`ftp` $\cup$ `http` $\cup$ `https` $\cup$ `mailto` $\cup$ `news`) : (// (($\varepsilon \cup$ <*Authority*>) (<*Path-Absolute*> $\cup$ <*Path-Empty*>) $\cup$ <*Path-Rootless*>) ($\varepsilon \cup$ (? <*Query*>)) ($\varepsilon \cup \#$ <*Fragment*>)

(`ftp` $\cup$ `http` $\cup$ `https` $\cup$ `mailto` $\cup$ `news`) :
(// (($\varepsilon \cup$ (($\varepsilon \cup$ (<*User-info*> @)) <*Host*> ($\varepsilon \cup$ (: <*Port*>)))
(/ ($\varepsilon \cup$ (<*Segment*-1> (/ <*Segment*-1>)*)) $\cup$ <*Path-Empty*>))
$\cup$ <*Path-Rootless*>)
($\varepsilon \cup$ (? <*Query*>)) ($\varepsilon \cup \#$ <*Fragment*>)

12) Prove that each of the following grammars is correct:
a) The grammar, shown in Example 11.3, for the language PalEven.

We first prove that $L(G) \subseteq L$ (i.e, every element generated by G is in L). Note a string w is in L iff it is ε or it is of the form `a`x`a` or `b`x`b` for some $x \in L$. So the proof is straightforward if we let the loop invariant I, describing the working string st, be:

$(st = s S s^R : s \in \{\texttt{a}, \texttt{b}\}^*) \vee (st = s s^R : s \in \{\texttt{a}, \texttt{b}\}^*)$.

When $st = S$, the invariant is satisfied by the first disjunct. It is preserved by all three rules of the grammar. If it holds and $st \in \{\texttt{a}, \texttt{b}\}^*$, then $st \in L$.

We then prove that $L \subseteq L(G)$ (i.e, every element of L is generated by G). Note that any string $w \in L$ must either be ε (which is generated by G (since $S \Rightarrow \varepsilon$)), or it must be of the form `a`x`a` or w = `b`x`b` for some $x \in L$. Also notice that every string in L has even length. This suggests an induction proof on the length of the derived string:

- Base step: $\varepsilon \in L$ and $\varepsilon \in L(G)$.
- Induction hypothesis: Every string in L of length k can be generated by G.
- Prove that every string in L of length k +2 can also be generated. (We use k+2 here rather than the more usual k+1 because, in this case, all strings in L have even length. Thus if a string in L has length k, there are no strings in L of length k +1.). If $|w| = k$+2 and $w \in L$, then w = `a`x`a` or w = `b`x`b` for some $x \in L$. $|x| = k$, so, by the induction hypothesis, $x \in L(G)$. Therefore $S \Rightarrow^* x$. So either $S \Rightarrow$ `a`S`a` $\Rightarrow^*$ `a`x`a`, and $x \in L(G)$, or $S \Rightarrow$ `b`S`b` $\Rightarrow^*$ `b`x`b`, and $x \in L(G)$.

b) The grammar, shown in Example 11.1, for the language Bal.

We first prove that $L(G) \subseteq$ Bal (i.e, every element generated by G is in Bal. The proof is straightforward if we let the loop invariant I, describing the working string st, be:

The parentheses in st are balanced.

When $st = S$, the invariant is trivially satisfied since there are no parentheses. It is preserved by all three rules of the grammar. If it holds and $st \in \{), (\}^*$, then $st \in$ Bal.

We then prove that Bal $\subseteq L(G)$ (i.e, every element of Bal is generated by G).

- Base step: $\varepsilon \in$ Bal and $\varepsilon \in L(G)$.
- Induction hypothesis: Every string in L of length k or less can be generated by G.
- Prove that every string in Bal of length k +2 can also be generated. (We use k+2 here rather than the more usual k+1 because, in this case, all strings in L have even length. Thus if a string in Bal has length k, there are no strings in Bal of length k +1.). To do this proof, we'll exploit the notion of siblings, as described in Example 11.21.

Let w be a string in Bal of length k+2. We can divide it into a set of siblings. To do this, find the first balanced set of parentheses that includes the first character of w. Then find that set's siblings. We consider two cases:

- There are no siblings (i.e., the opening left parenthesis isn't closed until the last character of w): Peel off the first and last character of w generating a new string w'. w' has length k. It is in Bal and so, by the induction hypothesis, can be generated by G. Thus w can be generated with the derivation that begins: $S \Rightarrow (S) \Rightarrow$ and then continues by expanding the remaining S to derive w'.
- There is at least one sibling. Then w and all its siblings are strings in Bal and each of them has length k or less. Thus, by the induction hypothesis, each of them can be generated by G. If w has n siblings, then it can be generated by the derivation that begins: $S \Rightarrow S\,S \Rightarrow S\,S\,S$ ..., applying the rule $S \rightarrow S\,S$ n times. Then the resulting S's can derive w' and its siblings, in order, thus deriving w.

13) For each of the following grammars G, show that G is ambiguous. Then find an equivalent grammar that is not ambiguous.

a) ($\{S, A, B, T, \mathtt{a}, \mathtt{c}\}$, $\{\mathtt{a}, \mathtt{c}\}$, R, S), where $R = \{S \rightarrow AB, S \rightarrow BA, A \rightarrow \mathtt{a}A, A \rightarrow \mathtt{ac}, B \rightarrow T\mathtt{c}, T \rightarrow \mathtt{a}T, T \rightarrow \mathtt{a}\}$.

Both A and B generate $\mathtt{a}^+\mathtt{c}$. So any string in L can be generated two ways. The first begins $S \Rightarrow AB$. The second begins $S \Rightarrow BA$. The easy fix is to eliminate one of A or B. We pick B to eliminate because it uses the more complicated path, through T. So we get: $G' = \{\{S, A, \mathtt{a}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{c}\}, R, S\}$, where $R = \{S \rightarrow AA, A \rightarrow \mathtt{a}A, A \rightarrow \mathtt{ac}\}$. G' is unambiguous. Any derivation in G' of the string $\mathtt{a}^n\mathtt{c}$ must be of the form: $S \Rightarrow AA \Rightarrow^{n-1} \mathtt{a}^{n-1}A \Rightarrow \mathtt{a}^{n-1}\mathtt{ac}$. So there is only one leftmost derivation in G' of any string in L.

b) ($\{S, \mathtt{a}, \mathtt{b}\}$, $\{\mathtt{a}, \mathtt{b}\}$, R, S), where $R = \{S \rightarrow \varepsilon, S \rightarrow \mathtt{a}S\mathtt{a}, S \rightarrow \mathtt{b}S\mathtt{b}, S \rightarrow \mathtt{a}S\mathtt{b}, S \rightarrow \mathtt{b}S\mathtt{a}, S \rightarrow SS\}$.

Note that $L(G) = \{w : w \in (\mathtt{a}, \mathtt{b})^*$ and $|w|$ is even$\}$. So we can just get rid of the rule $S \rightarrow SS$. Once we do that, the ε rule no longer causes ambiguity. So we have the rules: $\{S \rightarrow \varepsilon, S \rightarrow \mathtt{a}S\mathtt{a}, S \rightarrow \mathtt{b}S\mathtt{b}, S \rightarrow \mathtt{a}S\mathtt{b}, S \rightarrow \mathtt{b}S\mathtt{a}\}$.

c) ({*S*, *A*, *B*, *T*, a, c}, {a, c}, *R*, *S*), where $R = \{S \rightarrow AB, A \rightarrow AA, A \rightarrow \texttt{a}, B \rightarrow T\texttt{c}, T \rightarrow \texttt{a}T, T \rightarrow \texttt{a}\}$.

A splits into symmetric branches. So the derivation $A \Rightarrow AA \Rightarrow AAA$ already corresponds to two parse trees. We need to force branching in one direction or the other. We'll choose to branch to the left. So the new rule set is $R = \{S \rightarrow AB, A \rightarrow AA^*, A \rightarrow A^*, A^* \rightarrow \texttt{a}, B \rightarrow T\texttt{c}, T \rightarrow \texttt{a}T, T \rightarrow \texttt{a}\}$. Each A^* generates exactly one a. So, to generate a string with *n* a's, the rule $A \rightarrow AA^*$ must be applied exactly *n* times. Then the rule $A \rightarrow A^*$ must be applied.

d) ({*S*, a, b}, {a, b}, *R*, *S*), where $R = \{S \rightarrow \texttt{a}S\texttt{b}, S \rightarrow \texttt{b}S\texttt{a}, S \rightarrow SS\ , S \rightarrow \varepsilon\}$. (*G* is the grammar that we presented in Example 11.10 for the language $L = \{w \in \{\texttt{a},\texttt{b}\}^* : \#_{\texttt{a}}(w) = \#_{\texttt{b}}(w)\}$.)

$S \Rightarrow SS \Rightarrow \texttt{a}S\texttt{b}S \Rightarrow \texttt{aa}S\texttt{bb}S \Rightarrow \texttt{aabba}S\texttt{b} \Rightarrow \texttt{aabbab}$
$S \Rightarrow SS \Rightarrow SSS \Rightarrow \texttt{a}S\texttt{b}SS \Rightarrow \texttt{aa}S\texttt{bb}SS \Rightarrow \texttt{aabb}SS \Rightarrow \texttt{aabba}S\texttt{b}S \Rightarrow \texttt{aabbab}S \Rightarrow \texttt{aabbab}$
$S \Rightarrow \texttt{a}S\texttt{b} \Rightarrow \texttt{a}SS\texttt{b} \Rightarrow \texttt{aa}S\texttt{b}S\texttt{b} \Rightarrow \texttt{aab}S\texttt{b} \Rightarrow \texttt{aabb}S\texttt{ab} \Rightarrow \texttt{aabbab}$

Fixing this is not easy. We can try doing it the same way we did for Bal:

$S \rightarrow \varepsilon$
$S \rightarrow T$
$T \rightarrow T'T$
$T \rightarrow T'$
$T' \rightarrow \texttt{ab}$
$T' \rightarrow \texttt{a}T\texttt{b}$
$T' \rightarrow \texttt{ba}$
$T' \rightarrow \texttt{b}T\texttt{a}$

But we'll still get two parses for, say, aabbab: [aabb] [ab] and [a [ab] [ba] b]

To make an unambiguous grammar we will force that any character matches the closest one it can. So no nesting unless necessary.

$S \rightarrow \varepsilon$
$S \rightarrow T_a$ /* *A* region (which starts with a) first
$S \rightarrow T_b$ /* *B* region (which starts with b) first

$T_a \rightarrow A$ /* just a single *A* region
$T_a \rightarrow AB$ /* two regions, an *A* region followed by a *B* region
$T_a \rightarrow ABT_a$ /* more than two regions

$T_b \rightarrow B$ /* similarly if start with b
$T_b \rightarrow BA$
$T_b \rightarrow BAT_b$

$A \rightarrow A_1$ /* this *A* region can be just a single ab pair
$A \rightarrow A_1A$ /* or arbitrarily many balanced sets
$A_1 \rightarrow \texttt{a}A\texttt{b}$ /* a balanced set is a string of *A* regions inside a matching ab pair
$A_1 \rightarrow \texttt{ab}$

$B \rightarrow B_1$ /* similarly if start with b
$B \rightarrow B_1B$
$B_1 \rightarrow \texttt{b}B\texttt{a}$
$B_1 \rightarrow \texttt{ba}$

e) ({*S*, a, b}, {a, b}, *R*, *S*), where $R = \{S \rightarrow \mathtt{a}S\mathtt{b}, S \rightarrow \mathtt{aa}S\mathtt{b}, S \rightarrow \varepsilon\}$.

This grammar generates the language $\{\mathtt{a}^i\mathtt{b}^j : 0 \le j \le i \le 2j\}$. It generates two parse trees for the string aaabb. If can begin with either the first or the second *S* rule. An equivalent, unambiguous grammar is:

({*S*, *T*, a, b}, {a, b}, *R*, *S*), where $R = \{S \rightarrow \mathtt{a}S\mathtt{b}, S \rightarrow T, S \rightarrow \varepsilon, T \rightarrow \mathtt{aa}T\mathtt{b}, T \rightarrow \varepsilon\}$.

14) Let *G* be any context-free grammar. Show that the number of strings that have a derivation in *G* of length *n* or less, for any $n > 1$, is finite.

Define $L_n(G)$ to be the set of strings in $L(G)$ that have a derivation in *G* of length *n* or less. We can give a (weak) upper bound on the number of strings in $L_n(G)$. Let *p* be the number of rules in *G* and let *k* 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 *k* 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 no more than $p \cdot k \cdot p$. 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 *k* nonterminals that can be expanded and *p* ways to expand each. In general, an upper bound on the number of strings produced after *n* derivation steps is $p^n k^{(n-1)}$, 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.

15) Consider the fragment of a Java grammar that is presented in Example 11.20. How could it be changed to force each else clause to be attached to the outermost possible if statement?

```
<Statement> ::= <IfThenStatement> | <IfThenElseStatement>  | …
<StatementNoLongIf> ::= <block> | <IfThenStatement> | …          (But no <IfThenElseStatement> allowed here.)
<IfThenStatement> ::= if ( <Expression> ) <StatementNoLongIf>
<IfThenElseStatement> ::= if ( <Expression> ) <Statement> else <Statement>
```

16) How does the COND form in Lisp, as described in G.5, avoid the dangling else problem?

The COND form is a generalization of the standard If/then/else statement. It's more like a case statement since it can contain an arbitrary number of condition/action pairs. Every Lisp expression is a list, enclosed in parentheses. So, each action is delimited by parentheses. CONDs can be nested, but each complete conditional expression is delimited by parentheses. So there is no ambiguity about where any action belongs.

Example: (COND (<*condition* 1> <*action* 1>)
(<*condition* 2> (COND ((<*condition* 3>) (<*action* 3>))
((<*condition* 4>) (<*action* 4>))
)
)
)

17) Consider the grammar G' of Example 11.19.

a) Convert G' to Chomsky normal form.

Remove-units produces:

$E \to E + T \mid T * F \mid (E) \mid \texttt{id}$
$T \to T * F \mid (E) \mid \texttt{id}$
$F \to (E) \mid \texttt{id}$

Then the final result is:

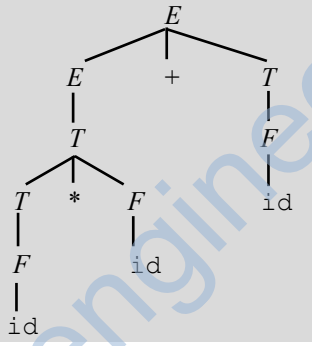
$E \to E\ E_1 \mid T\ E_2 \mid T_(\ E_3 \mid \texttt{id}$
$T \to T\ E_2 \mid T_(\ E_3 \mid \texttt{id}$
$F \to T_(\ E_3 \mid \texttt{id}$
$E_1 \to T_+\ T$
$E_2 \to T_*\ F$
$E_3 \to E\ T_)$
$T_(\to (\texttt{(}$
$T_) \to (\texttt{)}$
$T_+ \to (\texttt{+}$
$T_* \to (\texttt{*}$

b) Consider the string `id*id+id`.

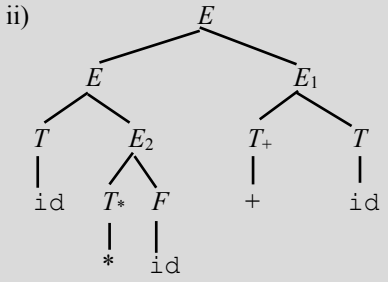
i) Show the parse tree that G' produces for it.

ii) Show the parse tree that your Chomsky normal form grammar produces for it.

i)



ii)



18) Convert each of the following grammars to Chomsky Normal Form:

a) $S \to \texttt{a}\ S\ \texttt{a}$
$S \to B$
$B \to \texttt{b b}\ C$
$B \to \texttt{b b}$
$C \to \varepsilon$
$C \to \texttt{c}\ C$

b) $S \to ABC$
$A \to \texttt{a}C \mid D$
$B \to \texttt{b}B \mid \varepsilon \mid A$
$C \to A\texttt{c} \mid \varepsilon \mid C\texttt{c}$
$D \to \texttt{aa}$

$S \rightarrow AS_1$
$S_1 \rightarrow BC$
$S \rightarrow AC$
$S \rightarrow AB$
$S \rightarrow X_aC \mid \mathtt{a} \mid X_aX_a$
$A \rightarrow X_aC$
$A \rightarrow \mathtt{a}$
$A \rightarrow X_aX_a$
$B \rightarrow X_bB$
$B \rightarrow \mathtt{b}$
$B \rightarrow \varepsilon$
$B \rightarrow X_aC \mid \mathtt{a} \mid X_aX_a$

$C \rightarrow AX_c$
$C \rightarrow \varepsilon$
$C \rightarrow CX_c$
$C \rightarrow \mathtt{c}$
$D \rightarrow X_aX_a$
$X_a \rightarrow \mathtt{a}$
$X_b \rightarrow \mathtt{b}$
$X_c \rightarrow \mathtt{c}$

c) $S \rightarrow \mathtt{a}TV\mathtt{a}$
$T \rightarrow \mathtt{a}T\mathtt{a} \mid \mathtt{b}T\mathtt{b} \mid \varepsilon \mid V$
$V \rightarrow \mathtt{c}V\mathtt{c} \mid \varepsilon$

$S \rightarrow AS_1 \mid AS_2 \mid AS_3 \mid AA$
$S_1 \rightarrow TS_2$
$S_2 \rightarrow VA$
$S_3 \rightarrow TA$
$T \rightarrow AA \mid BB \mid CC \mid CT_1 \mid AS_3 \mid BT_2$
$T_2 \rightarrow TB$

$V \rightarrow CT_1 \mid CC$
$T_1 \rightarrow VC$
$A \rightarrow \mathtt{a}$
$B \rightarrow \mathtt{b}$
$C \rightarrow \mathtt{c}$

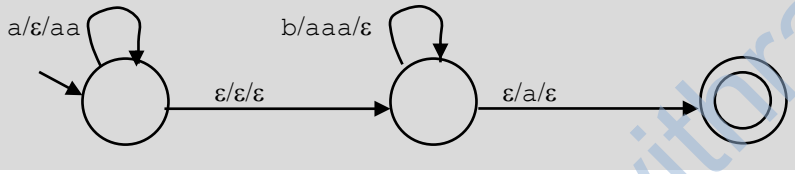
12 Pushdown Automata

1) Build a PDA to accept each of the following languages L:

a) BalDelim = {w : where w is a string of delimeters: (,), [,], {, }, that are properly balanced}.

$M = (\{1\}, \{(,), [,], \{, \}\}, \{(, [, \{\}, \Delta, 1, \{1\})$, where $\Delta =$
{ ((1, (, ε), (1, ())),
((1, [, ε), (1, [)),
((1, {, ε), (1, {)),
((1,),)), (1, ε)),
((1,],]), (1, ε)),
((1, }, }), (1, ε)) }

b) $\{a^i b^j : 2i = 3j + 1\}$.



c) $\{w \in \{a, b\}^* : \#_a(w) = 2 \cdot \#_b(w)\}$.

The idea is that we only need one state. The stack will do all the work. It will count whatever it is ahead on. Since one a matches two b's, each a will push an a (if the machine is counting a's) and each b (if the machine is counting a's) will pop two of them. If, on the other hand, the machine is counting b's, each b will push two b's and each a will pop one. The only tricky case arises with inputs like aba. M will start out counting a's and so it will push one onto the stack. Then comes a b. It wants to pop two a's, but there's only one. So it will pop that one and then switch to counting b's by pushing a single b. The final a will then pop that b. M is highly nondeterministic. But there will be an accepting path iff the input string w is in L.

$M = (\{1\}, \{a, b\}, \{a, b\}, \Delta, 1, \{1\})$, where $\Delta =$
{ ((1, a, ε), (1, a)),
((1, a, b), (1, ε)),
((1, b, ε), (1, bb)),
((1, b, aa), (1, ε)),
((1, b, a), (1, b)) }

d) $\{a^n b^m : m \le n \le 2m\}$.

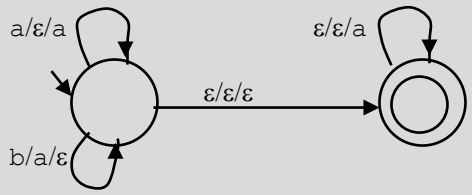
$M = (\{1, 2\}, \{a, b\}, \{a\}, \Delta, 1, \{1, 2\})$, where $\Delta =$
{ ((1, a, ε), (1, a)),
((1, ε, ε), (2, ε)),
((2, b, a), (2, ε)),
((2, b, aa), (2, ε)) }.

e) $\{w \in \{a, b\}^* : w = w^R\}$.

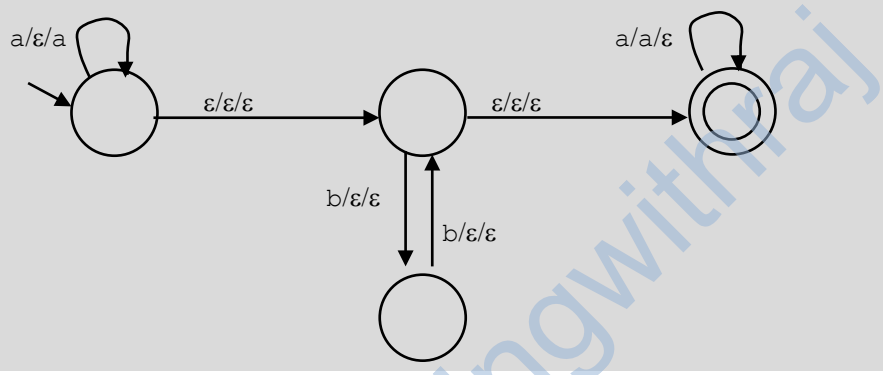
This language includes all the even-length palindromes of Example 12.5, plus the odd-length palindromes. So a PDA to accept it has a start state we'll call 1. There is a transition, from 1, labeled ε/ε/ε, to a copy of the PDA of Example 12.5. There is also a similarly labeled transition from 1 to a machine that is identical to the machine of Example 12.5 except that the transition from state s to state f has the following two labels: a/ε/ε and b/ε/ε. If an input string has a middle character, that character will drive the new machine through that transition.

f) $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j \text{ or } j \neq k)\}$.

g) $\{w \in \{a, b\}^* : \text{every prefix of } w \text{ has at least as many a's as b's}\}$.



h) $\{a^n b^m a^n : n, m \geq 0 \text{ and } m \text{ is even}\}$.



i) $\{xc^n : x \in \{a, b\}^*, \#_a(x) = n \text{ or } \#_b(x) = n\}$.

M will guess whether to count a's or b's. $M = (\{1, 2, 3, 4\}, \{a, b, c\}, \{c\}, \Delta, 1, \{1, 4\})$, where $\Delta =$
{ ((1, ε, ε), (2, ε)),
((1, ε, ε), (3, ε)),
((2, a, ε), (2, c)),
((2, b, ε), (2, ε)),
((2, ε, ε), (4, ε)),
((3, a, ε), (3, ε)),
((3, b, ε), (3, c)),
((3, ε, ε), (4, ε)),
((4, c, c), (4, ε)) }

j) $\{a^n b^m : m \geq n, m\text{-}n \text{ is even}\}$.

$M = (\{1, 2, 3, 4\}, \{a, b\}, \{a\}, \Delta, 1, \{3\})$, where $\Delta =$
{ ((1, a, ε), (1, a)),
((1, b, a), (2, ε)),
((1, ε, ε), (3, ε)),
((2, b, a), (2, ε)),
((2, ε, ε), (3, ε)),
((3, b, ε), (4, ε)),
((4, b, ε), (3, ε)) }

k) $\{a^m b^n c^p d^q : m, n, p, q \geq 0 \text{ and } m + n = p + q\}$.

l) $\{b_i \# b_{i+1}^R : b_i \text{ is the binary representation of some integer } i, i \geq 0, \text{ without leading zeros}\}$. (Example: 101#011 ∈ *L*.)

m) $\{x^R\#y : x, y \in \{0, 1\}^* \text{ and } x \text{ is a substring of } y\}$.

n) L_1^*, where $L_1 = \{xx^R : x \in \{a,b\}^*\}$.

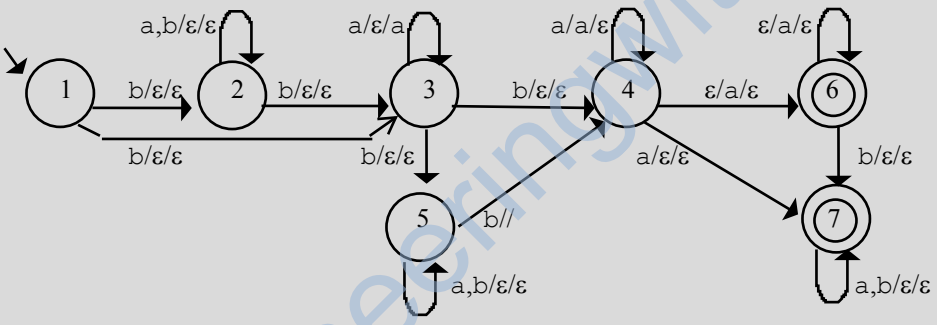
$M = (\{1, 2, 3, 4\}, \{a, b\}, \{a, b, \#\}, \Delta, 1, \{4\})$, where $\Delta =$

{ ((1, ε, ε), (2, #)),
((2, a, ε), (2, a)),
((2, b, ε), (2, b)),
((2, ε, ε), (3, ε)),
((3, a, a), (3, ε)),
((3, b, b), (3, ε)),
((3, ε, #), (4, ε)),
((3, ε, #), (2, #)) }

2) Complete the PDA that we sketched, in Example 12.8, for $\neg A^nB^nC^n$, where $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$.

3) Let $L = \{ba^{m1}ba^{m2}ba^{m1=3}\ldots ba^{mn} : n \geq 2, m_1, m_2, \ldots, m_n \geq 0, \text{ and } m_i \neq m_j \text{ for some } i, j\}$.

a) Show a PDA that accepts L .



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) Show a context-free grammar that generates L.

$S \rightarrow A'baLA'$	/* L will take care of two groups where the first group has more a's
$S \rightarrow A'baRA'$	/* R will take care of two groups where the second group has more a's
$L \rightarrow aA'b \mid aL \mid aLa$	
$R \rightarrow bA'a \mid Ra \mid aRa$	
$A' \rightarrow bAA' \mid \varepsilon$	/* generates 0 or more ba* strings
$A \rightarrow aA \mid \varepsilon$	

c) Prove that L is not regular.

Let $L_1 = ba^*ba^*$, which is regular because it can be described by a regular expression. 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. So we complete the proof by doing that.

4) Consider the language $L = L_1 \cap L_2$, where $L_1 = \{ww^R : w \in \{a, b\}^* \text{ and } L_2 = \{a^nb^*a^n : n \geq 0\}$.

a) List the first four strings in the lexicographic enumeration of L?

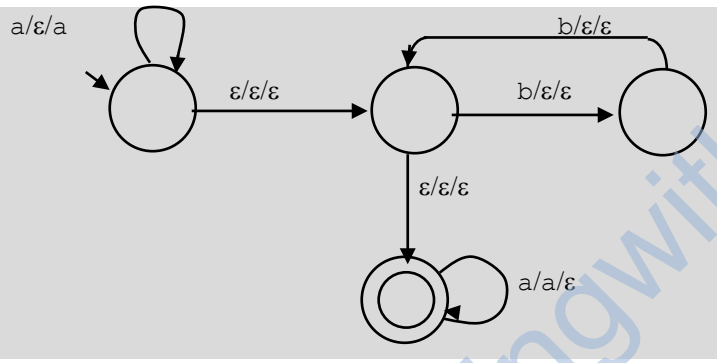
ε, aa, bb, aaaa

b) Write a context-free grammar to generate L.

Note that $L = \{a^n b^{2m} a^n : n, m \geq 0\}$.

$S \rightarrow aSa$
$S \rightarrow B$
$B \rightarrow bBb$
$B \rightarrow \varepsilon$

c) Show a natural pda for L. (In other words, don't just build it from the grammar using one of the two-state constructions presented in the book.)



d) Prove that L is not regular.

Note that $L = \{a^n b^{2m} a^n : n, m \geq 0\}$. We can prove that it is not regular using the Pumping Theorem. Let $w = a^k b^{2k} a^k$. Then y must fall in the first a region. Pump in once. The number of a's in the first a region no longer equals the number of a's in the second a region. So the resulting string is not in L. L is not regular.

5) Build a deterministic PDA to accept each of the following languages:

a) $L\$$, where $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$.

The idea is to use a bottom of stack marker so that M can tell what it should be counting. If the top of the stack is an a, it is counting a's. If the top of the stack is a b, it is counting b's. If the top of the stack is #, then it isn't counting anything yet. So if it is reading an a, it should start counting a's. If it is reading a b, it should start counting b's.

$M = (\{0, 1, 2\}, \{a, b\}, \{a, b\}, 0, \{2\}, \Delta)$, where $\Delta =$

{((0, ε, ε), (1, #)),	((1, $, #),(2, ε)),	/* starting and ending.
((1, a, a), (1, aa)),	((1, b, a), (1, ε)),	/* already counting a's.
((1, a, b), (1, ε)),	((1, b, b), (1, bb)),	/* already counting b's.
((1, a, #), (1, a#)),	((1, b, #), (1, b#))}	/* not yet counting anything. Start now.

b) $L\$$, where $L = \{a^n b^+ a^m : n \geq 0$ and $\exists k \geq 0\ (m = 2k+n)\}$.

The number of a's in the second a region must equal the number in the first region plus some even number. M will work as follows: It will start by pushing # onto the stack as a bottom marker. In the first a region it will push one a for each a it reads. Then it will simply read all the b's without changing the stack. Then it will pop one a for each a it reads. When the # becomes the top of the stack again, unless $ appears at the same time, M will become a simple DFSM that has two states and checks that there is an even number of a's left to read. When it reads the $, it halts (and accepts).

$M = (\{1, 2, 3, 4, 5, 6, 7\}, \{a, b\}, \{a\}, 1, \{7\}, \Delta)$, where $\Delta =$
{((1, ε, ε), (2, #)),
((2, a, ε), (2, a)), ((2, b, ε), (3, ε)),
((3, b, ε), (3, ε)), ((3, a, a), (4, ε)),
((4, a, a), (4, ε)), ((4, $, #), (7, ε)), ((4, a, #), (6, ε)),
((5, a, ε), (6, ε)), ((5, $, ε), (7, ε)),
((6, a, ε), (5, ε)) }

6) Complete the proof that we started in Example 12.14. Specifically, show that if, M is a PDA that accepts by accepting state alone, then there exists a PDA M' that accepts by accepting state and empty stack (our definition) where $L(M') = L(M)$.

By construction: We build a new PDA P' from P as follows: Let P' initially be P. Add to P' a new accepting state F. From every original accepting state in P', add an epsilon transition to F. Make F the only accepting state in P'. For every element g of Γ, add the following transition to P': $((F, \varepsilon, g), (F, \varepsilon))$. In other words, if and only if P accepts, go to the only accepting state of P' and clear the stack. Thus P' will accept by accepting state and empty stack iff P accepts by accepting state.

13 Context-Free and Noncontext-Free Languages

1) For each of the following languages L, state whether L is regular, context-free but not regular, or not context-free and prove your answer.

a) $\{xy : x, y \in \{\mathtt{a}, \mathtt{b}\}^* \text{ and } |x| = |y|\}$.

Regular. $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : |w| \text{ is even}\}$
$= (\mathtt{aa} \cup \mathtt{ab} \cup \mathtt{ba} \cup \mathtt{bb})^*$

b) $\{(\mathtt{ab})^n \mathtt{a}^m \mathtt{b}^n : n > 0\}$.

Not context-free. We prove this with the Pumping Theorem. Let $w = (\mathtt{ab})^k \mathtt{a}^k \mathtt{b}^k$. Divide w into three regions: the ab region, the a region, and the b region. If either v or y crosses the boundary between regions 2 and 3 then pump in once. The resulting string will have characters out of order. We consider the remaining alternatives for where nonempty v and y can occur:
(1, 1) If $|vy|$ is odd, pump in once and the resulting string will have characters out of order. If it is even, pump in once. The number of ab's will no longer match the number of a's in region 2 or b's in region 3.
(2, 2) Pump in once. More a's in region 2 than b's in region 3.
(3, 3) Pump in once. More b's in region 3 than a's in region 2.
v or y crosses the boundary between 1 and 2: Pump in once. Even if v and y are arranged such that the characters are not out of order, there will be more ab pairs than there are b's in region 3.
(1, 3) $|vxy|$ must be less than or equal to k.

c) $\{x\#y : x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } x \neq y\}$.

Context-free not regular. We can build a PDA M to accept L. All M has to do is to find one way in which x and y differ. We sketch its construction: M starts by pushing a bottom of stack marker Z onto the stack. Then it nondeterministically chooses to go to state 1 or 2. From state 1, it pushes the characters of x, then starts popping the characters of y. It accepts if the two strings are of different lengths. From state 2, it must accept if two equal length strings have at least one different character. So M starts pushing a % for each character it sees. It nondeterministically chooses a character on which to stop. It remembers that character in its state (so it branches and there are two similar branches from here on). It reads the characters up to the # and does nothing with them. Starting with the first character after the #, it pops one % for each character it reads. When the stack is empty it checks to see whether the next input character matches the remembered character. If it does not, it accepts.

d) $\{\mathtt{a}^i \mathtt{b}^n : i, n > 0 \text{ and } i = n \text{ or } i = 2n\}$.

Context-free, not regular. L can be generated by the following context-free grammar:

$S \rightarrow A \mid B$
$A \rightarrow \mathtt{a}A\mathtt{b} \mid \mathtt{ab}$
$B \rightarrow \mathtt{aa}B\mathtt{b} \mid \mathtt{aab}$

L is not regular, which we show by pumping: Let $w = \mathtt{a}^k \mathtt{b}^k$. Pump out. To get a string in L, i must be equal to n or greater than n (in the case where $i = 2n$). Since we started with $i = n$ and then decreased i, the resulting string must not be in L.

e) $\{wx : |w| = 2 \cdot |x|$ and $w \in \mathtt{a}^+\mathtt{b}^+$ and $x \in \mathtt{a}^+\mathtt{b}^+\}$.

Context-free, not regular. L can be accepted by a PDA M that pushes one character for each a and b in w and then pops two characters for each a and b in x.

L is not regular, which we show by pumping. Note that the boundary between the w region and the x region is fixed; it's immediately after the last b in the first group. Choose to pump the string $w = \mathtt{a}^{2k}\mathtt{b}^{2k}\mathtt{a}^k\mathtt{b}^k$. $y = \mathtt{a}^p$, for some nonzero p. Pump in or out. The length of w changes but the length of x does not. So the resulting string is not in L.

f) $\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^k : n, m, k \geq 0$ and $m \leq min(n, k)\}$.

Not context-free. We prove it using the Pumping Theorem. Let k be the constant from the Pumping Theorem and let $w = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$. Let region 1 contain all the a's, region 2 contain all the b's, and region 3 contain all the c's. If either v or y crosses numbered regions, pump in once. The resulting string will not be in L because it will violate the form constraint. We consider the remaining cases for where nonempty v and y can occur:

(1, 1): Pump out once. This reduces the number of a's and thus the *min* of the number of a's and c's. But the number of b's is unchanged so it is greater than that minimum.
(2, 2): Pump in once. The *min* of the number of a's and c's is unchanged. But the number of b's is increased and so it is greater than that minimum.
(3, 3): Same argument as (1, 1) but reduces the number of c's.
(1, 2), (2, 3): Pump in once. The *min* of the number of a's and c's is unchanged. But the number of b's is increased and so it is greater than that minimum.
(1, 3): Not possible since $|vxy|$ must be less than or equal to k.

g) $\{xyx^R : x \in \{0, 1\}^+$ and $y \in \{0, 1\}^*\}$.

Regular. There is no reason to let x be more than one character. So all that is required is that the string have at least two characters and the first and last must be the same. $L = (0\ (0 \cup 1)^*\ 0) \cup (1\ (0 \cup 1)^*\ 1)$.

h) $\{xwx^R : x, w \in \{\mathtt{a}, \mathtt{b}\}^+$ and $|x| = |w|\}$.

Not context-free. If L were context-free, then $L_1 = L \cap \mathtt{a}^*\mathtt{b}^*\mathtt{a}^*\mathtt{a}\mathtt{b}^*\mathtt{a}^*$ would also be context-free. But we show that it is not by pumping. Let $w =$

$$\underset{1}{\mathtt{a}^{k+1}}\ \underset{2}{\mathtt{b}^{k+1}}\ \Big|\ \underset{3}{\mathtt{a}^{k+1}}\ \underset{4}{\mathtt{b}^{k}}\ \underset{5}{\mathtt{a}}\ \Big|\ \underset{6}{\mathtt{b}^{k+1}}\ \underset{7}{\mathtt{a}^{k+1}}$$

We break w into regions as shown above. If either v or y from the pumping theorem crosses numbered regions, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. If either v or y includes region 5, pump in once and the resulting string will not be in L_1 because it will violate the form constraint. If $|vy|$ is not divisible by 3, pump in once. The resulting string will not be able to be cut into three pieces of equal length and so it is not in L_1. We now consider the other ways in which nonempty v and y could occur:
(1, 1), (2, 2), (1, 2): Pump in twice. One a and at least one b move into the last third of the string. So the first third is $\mathtt{a}^p\mathtt{b}^q$, for some p and q, while the last third is $\mathtt{b}^i\mathtt{a}^j\mathtt{b}^k$, for some i, j, and k. So the last third is not the reverse of the first third.
(6. 6), (7, 7), (6, 7): Pump in twice. At least two a's will move into the first third of the string. So the first third is $\mathtt{a}^p\mathtt{b}^q\mathtt{a}^r$, for some p, q, and r, while the last third is $\mathtt{a}^j\mathtt{b}^k$, for some j and k. So the last third is not the reverse of the first third.

(3, 3), (4, 4), (3, 4): Pump in twice. At least two a's will move into the first third of the string and one a and at least one b will move into the last third. So the first third is $a^p b^q a^r$, for some p, q, and r, while the last third is $b^i a^j b^k$, for some i, j, and k. So the last third is not the reverse of the first third.
(2, 3): One a and at least one b will move into the last third of the string, so it will be $b^i a^j b^k$, for some i, j, and k. Depending on the length of v relative to the length of y, the first third will be either $a^p b^q a^r$, for some p, q, and r or $a^p b^q$, for some p and q. In either case, the last third is not the reverse of the first third.
(4, 6): Pump in twice. At least two a's will move into the first third of the string. So the first third is $a^p b^q a^r$, for some p, q, and r. Depending on the length of v relative to the length of y, the last third will be $a^j b^k$, for some j and k or $b^i a^j b^k$, for some i, j, and k. In either case, the last third is not the reverse of the first third.
All remaining cases are impossible since $|vxy|$ must be less than or equal to k.

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

Not context free. We prove it using the Pumping Theorem. Let $w = a^k b^k b^k a^k a^k b^k$.
1 | 2 | 3 | 4
In each of these cases, pump in once:

- If any part of v is in region 1, then to produce a string in L we must also pump a's into region 3. But we cannot since $|vxy| \le k$.
- If any part of v is in region 2, then to produce a string in L we must also pump b's into region 4. But we cannot since $|vxy| \le k$.
- If any part of v is in region 3, then to produce a string in L we must also pump a's into region 1. But we cannot since y must come after v.
- If any part of v is in region 4, then to produce a string in L we must also pump b's into region 2. But we cannot since y must come after v.

j) $\{wxw: |w| = 2 \cdot |x|$ and $w \in \{a, b\}^*$ and $x \in \{c\}^*\}$.

Not context-free. We prove it using the Pumping Theorem. Let $w = a^{2k} c^k a^{2k}$.

k) $\{a^i: i \ge 0\}$ $\{b^i: i \ge 0\}$ $\{a^i: i \ge 0\}$.

Regular. $L = a^*b^*c^*$.

l) $\{x \in \{a, b\}^* : |x|$ is even and the first half of x has one more a than does the second half$\}$.

Not context-free. In a nutshell, we can use the stack either to count the a's or to find the middle, but not both.

If L were context-free, then $L' = L \cap a^*b^*a^*b^*$ would also be context-free. But it is not, which we show using the Pumping Theorem. Let $w = a^{k+1} b^k a^k b^{k+1}$. Divide w into four regions (a's, then b's, then a's, then b's). If either v or y crosses a region boundary, pump in. The resulting string is not in L because the characters will be out of order. If $|vy|$ is not even, pump in once. The resulting string will not be in L because it will have odd length. Now we consider all the cases in which neither v nor y crosses regions and $|vy|$ is even:
(1, 1): pump in once. The boundary between the first half and the second half shifts to the left. But, since $|vxy| \le k$, only b's flow into the second half. So we added a's to the first half but not the second.
(2, 2): pump out. Since $|vy|$ is even, we pump out at least 2 b's so at least one a migrates from the second half to the first.
(3, 3): pump out. This decreases the number of a's in the second half. Only b's migrate in from the first half.
(4, 4): pump in once. The boundary between the first half and the second half shifts to the right, causing a's to flow from the second half into the first half.

(1, 2): pump in once. The boundary shifts to the left, but only b's flow to the second half. So a's were added in the first half but not the second.
(2, 3): pump out. If $|v| < |y|$ then the boundary shifts to the left. But only b's flow. So the number of a's in the second half is decreased but not the number of a's in the first half. If $|y| < |v|$ then the boundary shifts to the right. a's are pumped out of the second half and some of them flow into the first half. So the number of a's in the first half increases and the number of a's in the second half decreases. If $|v| = |y|$, then a's are pumped out of the second half but not the first.
(3, 4): pump out. That means pumping a's out of the second half and only b's flow in from the first half.
(1, 3), (1, 4), (2, 4): $|vxy|$ must be less than or equal to k.

m) $\{w \in \{a, b\}^* : \#_a(w) = \#_b(w)$ and w does not contain either the substring aaa or abab$\}$.

Context-free not regular. $L = L_1 \cap L_2$, where:

- $L_1 = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$, and
- $L_2 = \{w \in \{a, b\}^* : w$ does not contain either the substring aaa or abab$\}$

In the book we showed that L_1 is context-free. L_2 is regular. (There is a simple FSM that accepts it.) So L is the intersection of a context-free language and a regular language. So it must be context-free.

n) $\{a^n b^{2n} c^m\} \cap \{a^n b^m c^{2m}\}$.

Not context free. $L = \{a^n b^{2n} c^{4n}\}$. We prove L is not context free by pumping. Let $w = a^k b^{2k} c^{4k}$.
1 | 2 | 3
If either v or y contains more than a single letter, then pump in once and the resulting string will fail to be in L because it will have letters out of order. So each must fall within a single region:
(1, 1): pump in once. The number of b's will not be twice the number of a's.
(2, 2): pump in once. The number of b's will not be twice the number of a's.
(3, 3): pump in once. The number of c's will not be 4 times the number of a's.
(1, 2): pump in once. The number of c's will not change and so it will not be twice the number of b's.
(2, 3): pump in once. The number of a's will not change and so the number of c's will not be 4 times the number of a's.
(1, 3): cannot happen since $|vxy| \le k$.

o) $\{x \text{ c } y : x, y \in \{0, 1\}^*$ and y is a prefix of $x\}$.

Not context-free. If L were context-free, then $L_1 = L \cap 0^*1^*c0^*1^*$ would also be context-free. But we show that it is not by pumping. Let $w = 0^k 1^k$ c $0^k 1^k$. w is in L_1.
| 1 | 2 |3 |4 | 5

We break w into regions as shown above. If either v or y from the Pumping Theorem crosses numbered regions, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. If either v or y contains region 3, pump in once and the form constraint will be violated. We now consider the other ways in which nonempty v and y could occur:
(1, 1), (2, 2), (1, 2) Pump out. The region after the c is too long to be a prefix of the region before the c.
(4, 4), (5,5), (4, 5) Pump in. The region after the c is too long to be a prefix of the region before the c.
(2, 4) Pump out. The region after the c is no longer a prefix of the region before the c.
All other possibilities are ruled out by the requirement that $|vxy|$ be $\le k$.

p) $\{w : w = uu^R$ or $w = ua^n : n = |u|, u \in \{a, b\}^*\}$.

Context Free. $L = L_1 \cup L_2$, where $L_1 = uu^R$ and $L_2 = ua^n$, $n = |u|$. L_1 is CF because the following grammar generates it:

$S \rightarrow \varepsilon \mid aSa \mid bSb$

L_2 is CF because the following grammar generates it:

$S \rightarrow \varepsilon \mid aSa \mid bSa$

Since the CF languages are closed under union, L must be CF.

L is not regular, which we show by pumping. Let $w = b^k a^k a^k b^k$. Since $|xy| \le k$, y must occur in the first b region. When we pump in, we add b's to the first b region, but not the second one, so the resulting string is not in uu^R. It is also not in ua^n since it does not end in a's. So it is not in L .

q) $L(G)$, where $G = S \rightarrow aSa$

$S \rightarrow SS$

$S \rightarrow \varepsilon$

Regular. $L = (aa)^*$.

r) $\{w \in (A\text{-}Z, a\text{-}z, ., \text{blank})^+$: there exists at least one duplicated, capitalized word in $w\}$. For example, the sentence, The history of China can be viewed from the perspective of an outsider or of someone living in China $\in L$.

Not context-free. If L is context-free, so is $L' = L \cap Aa^*b^*\ Aa^*b^*$. We exploit this intersection property in order to make it impossible to pump into any blank region of w.

We show that L' is not context-free by pumping. Let:

$w = Aa^k b^k\ \ Aa^k b^k$

1| 2| 3|4|5| 6| 7

If any part of nonempty v or y goes into region 1, 4, or 5, pump in once and the resulting string violates the form constraint of L'. If any part of v or y crosses the 2/3 or 6/7 boundary, pump in once and the resulting string violates the form constraint of L'. We consider the remaining cases for where nonempty v and y can occur:

(2, 2), (2, 3), (3, 3) : pump in or out and the first word is changed but not the second one.
(5, 5), (5, 6), (6, 6) : pump in or out and the second word is changed but not the first one.
(3, 5) : pump in or out and the two words are changed in different ways.
(2, 6), (2, 7), (3, 7) : $|vxy| \le k$.

s) $\neg L_0$, where $L_0 = \{ww : w \in \{\mathtt{a}, \mathtt{b}\}^*\}$.

Context-free, not regular. Observe that L includes all strings in $\{\mathtt{a}, \mathtt{b}\}^*$ of odd length. It also includes all even length strings of the form $x\mathtt{a}y\,z\mathtt{b}q$, where $|x| = |z|$ and $|y| = |q|$. In other words there is some position in the first half that contains an $\mathtt{a}$, while the corresponding position in the second half contains $\mathtt{b}$, or, similarly, where there's a $\mathtt{b}$ in the first half and an $\mathtt{a}$ in the second half. Those strings look like $x\mathtt{b}y\,z\mathtt{a}q$. The only thing that matters about yz is its length and we observe that $|yz| = |zy|$. So an alternative way to describe the strings of the form $x\mathtt{a}y\,z\mathtt{b}q$ is $x\mathtt{a}z\,y\mathtt{b}q$, where $|x| = |z|$ and $|y| = |q|$. With this insight, we realize that $L =$

$\{w \in \{\mathtt{a}, \mathtt{b}\}^* : |w| \text{ is odd}\} \cup$ [1]

$\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w = x\mathtt{a}z\,y\mathtt{b}q$, where $|x| = |z|$ and $|y| = |q|\} \cup$ [2]

$\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w = x\mathtt{b}z\,y\mathtt{a}q$, where $|x| = |z|$ and $|y| = |q|\}$ [3]

Now we can build either a grammar or a PDA for each piece.

The PDAs for [2] and [3] work by pushing a symbol onto the stack for each symbol in x, then guessing at the symbol to remember. Then they pop a symbol for each input symbol until the stack is empty. Then they push one for every input symbol until they guess that they're at the mismatched symbol. After reading it, they pop one from the stack for every input symbol. If the stack and the input run out at the same time, the second guess chose the matching position. If the two characters that are checked are different, accept, else reject.

A grammar for [2] is: $S \rightarrow S_\mathtt{a}S_\mathtt{b}$, $S_\mathtt{a} \rightarrow CS_\mathtt{a}C$, $S_\mathtt{a} \rightarrow \mathtt{a}$, $S_\mathtt{b} \rightarrow CS_\mathtt{b}C$, $S_\mathtt{b} \rightarrow \mathtt{b}$, $C \rightarrow \mathtt{a}$, $C \rightarrow \mathtt{b}$. The grammar for [3] is analogous except that its S rule is $S \rightarrow S_\mathtt{b}S_\mathtt{a}$.

t) L^*, where $L = \{\mathtt{0}^*\mathtt{1}^i\mathtt{0}^*\mathtt{1}^i\mathtt{0}^*\}$.

Regular. While L is context free but not regular, L^* is regular. It's just the language of $\mathtt{1}$'s and $\mathtt{0}$'s where there is an even number of $\mathtt{1}$'s. The regular expression for L^* is $\mathtt{0}^*(\mathtt{10}^*\mathtt{10}^*)^*$.

u) $\neg A^nB^n$.

Context-free, not regular. The easy way to prove that L is context-free is to observe that A^nB^n is deterministic context-free and the deterministic context-free languages are closed under complement. It can also be done with a nondeterministic PDA. One branch will check for letters out of order. The other will check for mismatched numbers of $\mathtt{a}$'s and $\mathtt{b}$'s.

L is not regular because, if it were, then A^nB^n would also be regular since the regular languages are closed under complement. But we have shown that it isn't.

v) $\{\mathtt{b}\mathtt{a}^j\mathtt{b} : j = n^2$ for some $n \geq 0\}$. For example, $\mathtt{baaaab} \in L$.

Not context-free. We prove it using the Pumping Theorem. Let $n = k$, so $w = \mathtt{b}\mathtt{a}^r\mathtt{b}$, where $r = k^2$. If either v or y contains $\mathtt{b}$, pump out and the resulting string will not be in L. So $vy = \mathtt{a}^p$, for some p, where $0 < p \leq k$. After w, the next longest string in L is formed by letting $n = k + 1$, which yields a string with $(k + 1)^2$ $\mathtt{a}$'s. So the number of $\mathtt{a}$'s is $k^2 + 2k + 1$. But, if we start with w and pump in once, the longest possible string (which results if $|vy| = k$), has only $k^2 + k$ $\mathtt{a}$'s. It's too short to be in L. So L is not context-free.

w) $\{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}^* : \#_\mathtt{b}(w) \geq \#_\mathtt{c}(w) \geq \#_\mathtt{d}(w) \geq 0\}$.

Not context-free. If L were context-free, then $L_1 = L \cap \mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ would also be context free. But we show that it is not by pumping.

Let $w = \mathtt{b}^k\,\mathtt{c}^k\,\mathtt{d}^k$.

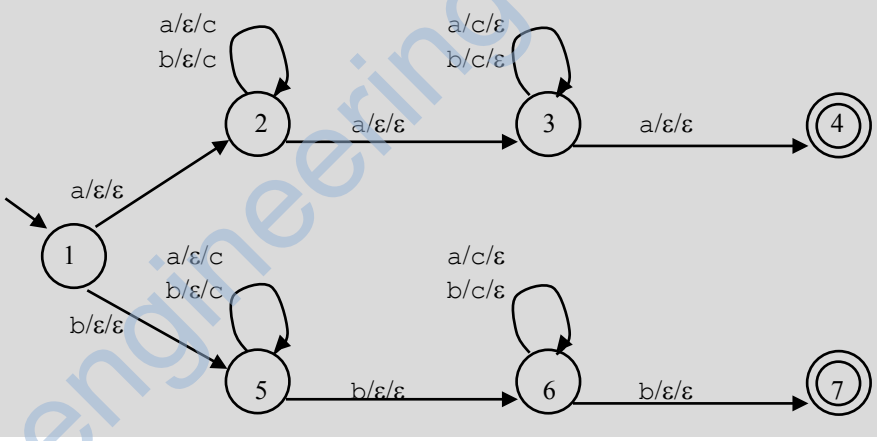
1 | 2 | 3

If either v or y from the pumping theorem contains two or more distinct letters, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. We consider the remaining cases for where nonempty v and y can occur:
(1, 1) Pump out. Fewer b's than c's.
(2, 2) Pump out. Fewer c's than d's.
(3, 3) Pump in. More d's than c's.
(1, 2) Pump out. Fewer b's than d's.
(2, 3) Pump in. More d's than a's.
(1, 3) $|vxy|$ must be less than k.

2) Let $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* :$ the first, middle, and last characters of w are identical$\}$.
   a) Show a context-free grammar for L.

$S \rightarrow A \mid B$
$A \rightarrow \mathtt{a}\ M_A\ \mathtt{a}$
$M_A \rightarrow CM_AC \mid \mathtt{a}$
$B \rightarrow \mathtt{b}\ M_B\ \mathtt{b}$
$M_B \rightarrow CM_BC \mid \mathtt{b}$
$C \rightarrow \mathtt{a} \mid \mathtt{b}$

   b) Show a natural PDA that accepts L.



   c) Prove that L is not regular.

If L is regular, then so is $L' = L \cap \mathtt{ab}^*\mathtt{ab}^*\mathtt{a}$. But we show that it is not. Let $w = \mathtt{ab}^k\mathtt{ab}^k\mathtt{a}$.
1| 2|3| 4|5

If y crosses numbered regions, pump in once. The resulting string will not be in L' because the letters will be out of order. If y is in region 1, 3, or 5, pump out and the resulting string will not be in L'. Because y must occur within the first k characters, the only place it can fall is within region 2. Pump in once. If $|y|$ is odd, the resulting string is not in L' because it has no middle character. If $|y|$ is even and thus at least 2, the resulting string is not in L' because its middle character is b, yet its first and last characters are a's.

3) Let $L = \{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^n\mathtt{d}^m : n, m \geq 1\}$. L is interesting because of its similarity to a useful fragment of a typical programming language in which one must declare procedures before they can be invoked. The procedure declarations include a list of the formal parameters. So now imagine that the characters in $\mathtt{a}^n$ correspond to the formal parameter list in the declaration of procedure 1. The characters in $\mathtt{b}^m$ correspond to the formal parameter

list in the declaration of procedure 2. Then the characters in c^n and d^m correspond to the parameter lists in an invocation of procedure 1 and procedure 2 respectively, with the requirement that the number of parameters in the invocations match the number of parameters in the declarations. Show that L is not context-free.

Not context-free. We prove it using the Pumping Theorem. Let $w = a^k b^k c^k d^k$.
1 | 2 | 3 | 4

If either v or y from the pumping theorem crosses numbered regions, pump in once. The resulting string will not be in L because the letters will be out of order. We now consider the other ways in which nonempty v and y could occur:
(1, 1), (3, 3) Pump in once. The a and c regions will no longer be of equal length.
(2, 2), (4, 4) Pump in once. The b and d regions will no longer be of equal length.
(1, 2), (3, 4), (2, 3) Pump in once. Neither the a and c regions nor the b and d regions will be of equal length.
(1, 3), (1, 4), (2, 4) Not possible since $|vxy| \le k$.
So L is not context-free.

4) Without using the Pumping Theorem, prove that $L = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)$ and $\#_a(w) > 50\}$ is not context-free.

Let $L_1 = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)$ and $\#_a(w) \le 50\}$. L_1 is regular, and thus also context-free, because it is finite. Suppose L were context-free. Then $L_2 = L \cup L_1$ would also be context-free, since the context-free languages are closed under union. But $L_2 = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}$,which we have shown is not context-free.

5) Give an example of a context-free language L ($\neq \Sigma^*$) that contains a subset L_1 that is not context-free. Prove that L is context free. Describe L_1 and prove that it is not context-free.

Let $L = \{a^n b^m c^p : n \ge 0, m \ge 0, p \ge 0$, and $n = m$ or $m = p\}$. L is context free because 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). $L_1 = \{a^n b^m c^p : n = m$ and $m = p\}$ is a subset of L. But $L_1 = A^n B^n C^n = \{a^n b^n c^n : n \ge 0\}$, which we have shown is not context free.

6) Let $L_1 = L_2 \cap L_3$.
a) Show values for L_1, L_2, and L_3, such that L_1 is context-free but neither L_2 nor L_3 is.

Let: $L_1 = \{a^n b^n : n \ge 0\}$.
$L_2 = \{a^n b^n c^j : j \le n\}$.
$L_3 = \{a^n b^n c^j : j = 0$ or $j > n\}$.

b) Show values for L_1, L_2, and L_3, such that L_2 is context-free but neither L_1 nor L_3 is.

Let: $L_2 = a^*$.
$L_1 = \{a^n : n$ is prime$\}$.
$L_3 = \{a^n : n$ is prime$\}$.

7) Give an example of a context-free language L, other than one of the ones in the book, where $\neg L$ is not context-free.

Let $L = \{a^i b^j c^k : i \ne j$ or $j \ne 2k\}$. L is context free. Another way to describe it is:

$L = \{a^i b^j c^k : i > j$ or $i < j$ or $j > 2k$ or $j < 2k\}$.

We can write a context-free grammar for L by writing a grammar for each of its sublanguages. So we have:

$S \rightarrow S_1C \mid S_2C \mid AS_3 \mid AS_4$
$S_1 \rightarrow \mathtt{a}S_1\mathtt{b} \mid \mathtt{a}S_1 \mid \mathtt{a}$ /* More a's than b's.
$S_2 \rightarrow \mathtt{a}S_2\mathtt{b} \mid S_1\mathtt{b} \mid \mathtt{b}$ /* More b's than a's.
$S_3 \rightarrow \mathtt{bb}S_4\mathtt{c} \mid \mathtt{b}S_4 \mid \mathtt{b}$ /* $j > 2k$.
$S_4 \rightarrow \mathtt{bb}S_4\mathtt{c} \mid \mathtt{b}S_4\mathtt{c} \mid S_4\mathtt{c} \mid \mathtt{c}$ /* $j < 2k$.
$A \rightarrow \mathtt{a}A \mid \varepsilon$
$C \rightarrow \mathtt{c}C \mid \varepsilon$

$\neg L = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ and } j = 2k\} \cup \{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* : \text{the letters are out of order}\}$. $\neg L$ is not context-free. If it were, then $L' = \neg L \cap \mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$ would also be context-free. But $L' = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ and } j = 2k\}$, which can easily be shown not be be context-free by using the Pumping Theorem, letting $w = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^{2k}$.

8) Theorem 13.7 tells us that the context-free languages are closed under intersection with the regular languages. Prove that the context-free languages are also closed under union with the regular languages.

Every regular language is also context-free and the context-free languages are closed under union.

9) Complete the proof that the context-free languages are not closed under *maxstring* by showing that $L = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : k \le i \text{ or } k \le j\}$ is context-free but *maxstring*(L) is not context-free.

L is context-free because the following context-free grammar generates it:

$S \rightarrow I \mid J$
$I \rightarrow \mathtt{a}\ I\ \mathtt{c} \mid \mathtt{a}\ I \mid B$
$B \rightarrow \mathtt{b}\ B \mid \varepsilon$
$J \rightarrow A\ J \mid J'$
$A \rightarrow \mathtt{a}\ A \mid \varepsilon$
$J' \rightarrow \mathtt{b}\ J'\ \mathtt{c} \mid \mathtt{b}\ J' \mid \varepsilon$

maxstring(L) = $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : k = max(i, j)\}$. We show that it is not context-free by pumping:

Let $w = \mathtt{a}^k\,\mathtt{b}^k\,\mathtt{c}^k$, where k is the constant from the Pumping Theorem.
1 2 3
If either v or y from the pumping theorem contains two or more distinct letters, pump in once. The resulting string will not be in *maxstring*(L) because it will violate the form constraint. We consider the remaining cases for where nonempty v and y can fall:
(1, 1): pump in once. $max(i, j)$ increased but k didn't.
(2, 2): pump in once. $max(i, j)$ increased but k didn't.
(3, 3): pump out. k decreased but $max(i, j)$ didn't.
(1, 2): pump in once. $max(i, j)$ increased but k didn't.
(2, 3): pump out. k decreased but $max(i, j)$ didn't.
(1, 3) $|vxy|$ must be less than k.

10) Use the Pumping Theorem to complete the proof, started in L.3.3, that English is not context-free if we make the assumption that subjects and verbs must match in a "respectively" construction.

11) In N.1.2, we give an example of a simple musical structure that cannot be described with a context-free grammar. Describe another one, based on some musical genre with which you are familiar. Define a sublanguage that captures exactly that phenomenon. In other words, ignore everything else about the music you are considering and describe a set of strings that meets the one requirement you are studying. Prove that your language is not context-free.

12) Define the leftmost maximal P subsequence m of a string w as follows:
- P must be a nonempty set of characters.
- A string s is a P subsequence of w iff s is a substring of w and s is composed entirely of characters in P. For example `1`, `0`, `10`, `01`, `11`, `011`, `101`, `111`, `1111`, and `1011` are {`0`, `1`} subsequences of `2312101121111`.
- Let S be the set of all P subsequences of w such that, for each element t of S, there is no P subsequence of w longer than t. In the example above, S = {`1111`, `1011`}.
- Then m is the leftmost (within w) element of S. In the example above, m = `1011`.

a) Let L = {$w \in$ {`0-9`}* : if y is the leftmost maximal {`0`, `1`} subsequence of w then $|y|$ is even}. Is L regular (but not context free), context free or neither? Prove your answer.

L is not context free. If L were context free, then $L' = L \cap$ `1*31*31*` would also be context free. We show that it is not by pumping.

Let $w = 1^{2k}31^{2k-1}31^{2k-1}$. The leftmost maximal {`0`, `1`} subsequence of w is the initial string of $2k$ `1`'s. It is the longest {`0`, `1`} subsequence and it is of even length.

Let LM01S mean "leftmost maximal {`0`, `1`} subsequence"

We will divide w into 5 regions as follows:

$$\underset{1}{1^{2k}}\,\Big|\,\underset{2}{3}\,\Big|\,\underset{3}{1^{2k-1}}\,\Big|\,\underset{4}{3}\,\Big|\,\underset{5}{1^{2k-1}}$$

Neither v nor y can contain regions 2 or 4. If either of them did, then we could pump in once and have more than the two `3`'s required for every string in L'. We consider all remaining cases for where v and y can fall:
(1, 1): $vy = 1^p$. Pump out, producing w'. If p =1, then the initial sequence of `1`'s is still the LM01S of w', but its length is odd, so w' is not in L'. If $p > 1$, then the second sequence of `1`'s is the LM01S of w', but its length is also odd, so w' is not in L'.

(3, 3): if $|vy|$ is even, then pump in once, producing w'. The second sequence of `1`'s is now the LM01S of w' since it just grew by at least two characters. But its length is still odd, so w' is not in L. If $|vy|$ is odd, then pump in twice, producing w'. The second sequence of `1`'s is now the LM01S of w'. But its length is odd (it started out odd and we pumped in an even number of `1`'s.) So w' is not in L.

(5, 5): same argument as (3, 3) except that we'll pump into the third region of `1`'s.

(1, 3): pump out once. By the same argument given for (1, 1), the resulting string w' is not in L:
If a single `1` is pumped out of region 1, it will still be the LM01S but will have odd length.
If more than than one `1` is pumped out of region 1, then region 5 (the third sequence of `1`'s) will be the LM01S of the resulting string w' because at least one `1` will be pumped out of the second region. But that region has odd length. So w' is not in L.

(1, 5): not possible because $|vxy|$ must be $\le k$.

(3, 5): if $|v| \ge |y|$ then the same argument as (3, 3) (because the second region of `1`'s will be the LM01S after pumping in). If $|y| > |v|$ then the same argument as (5, 5).

b) Let $L = \{w \in \{a, b, c\}^* :$ the leftmost maximal $\{a, b\}$ subsequence of w starts with $a\}$. Is L regular (but not context free), context free or neither? Prove your answer.

L is not context free. If L were context free, then $L' = L \cap a^*b^*cb^*$ would also be context free. We show that it is not by pumping.

Let $w = a^k b^k c b^k$. The leftmost maximal $\{a, b\}$ subsequence of w is the initial string of a's. The pumping proof is analogous to that of part a.

13) Are the context-free languages closed under each of the following functions? Prove your answer.

a) $chop(L) = \{w : \exists x \in L\ (x = x_1 c x_2 \wedge x_1 \in \Sigma_L^* \wedge x_2 \in \Sigma_L^* \wedge c \in \Sigma_L \wedge |x_1| = |x_2| \wedge w = x_1 x_2)\}$.

Not closed. We prove this by showing a counterexample. Let $L = \{a^n b^n c a^m b^m, n, m \geq 0\}$. L is context-free.

$chop(L) =$

$a^n b^n a^m b^m$ (in case, in the original string $n = m$)

$\cup$

$a^n b^{n-1} c a^m b^m$ (in case, in the original string, $n > m$)

$\cup$

$a^n b^n c a^{m-1} b^m$ (in case, in the original string, $n < m$)

We show that $chop(L)$ is not context free. First, note that if $chop(L)$ is context free then so is:

$L' = chop(L) \cap a^*b^*a^*b^*$. $L' = a^n b^n a^n b^n$.

We show that L' is not context free by pumping. Let $w = a^k b^k a^k b^k$. The rest is straightforward.

b) $mix(L) = \{w: x, y, z: (x \in L, x = yz, |y| = |z|, w = yz^R)\}$.

Not closed. We prove this by showing a counterexample. Let $L = \{(aa)^n (ba)^{3n}, n \geq 0\}$. L is context-free, since it can be generated by the grammar:

$S \rightarrow aaSbababa$

$S \rightarrow \varepsilon$

So every string in L is of the form $(aa)^n(ba)^n \mid (ba)^n(ba)^n$, with the middle marked with a |. $mix(L) = (aa)^n(ba)^n \mid (ab)^n(ab)^n = (aa)^n(ba)^n \mid (ab)^{2n}$. We show that this language is not context-free using the Pumping Theorem.

Let $w = (aa)^k(ba)^k\ (ab)^{2k}$

1 | 2 | 3

If either v or y crosses regions, pump in once and the resulting string will not have the correct form to be in $mix(L)$. If $|vy|$ is not even, pump in once, which will result in an odd length string. All strings in $mix(L)$ have even length. We consider the remaining cases for where nonempty v and y can occur:

(1, 1) Pump in. need 5 a's for every 3 b's. Too many a's.

(2, 2) Pump in. In every string in $mix(L)$, there's an instance of aa between the ba region and the ab region. There needs to be the same number of ba pairs before it as there are ab pairs after it. There are now more ba pairs.

(3, 3) Pump in. " except now more ab pairs.

(1, 2) Pump in. Same argument as (2, 2).

(2, 3) Pump in. In every string in $mix(L)$, there must be 3 a's after the first b for every 2 a's before it. There are now too many.

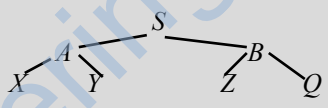
(1, 3) $|vxy| \leq k$.

c) $pref(L) = \{w: \exists x \in \Sigma^* (wx \in L)\}$.

Closed. We show this by construction. If L is context-free, then there exists some PDA M that accepts it. We construct M^* to accept $pref(L)$ as follows:
Initially, let $M^* = M$. Then create a new machine M' that is identical to M except that, for each transition labeled $x/y/z$ where x is an element of Σ, replace it with a transition labeled $\varepsilon/y/z$. Note that M' will mimic all the paths that M could follow but without consuming any input. In particular, it will perform all the same stack operations that M would do. Thus M' could finish any computation M could have done but without the required input. M' will accept if there is a path to one of its accepting states that also clears the stack. So, finally, add M' to M^* by connecting each state of M^* to its corresponding state in M' with a transition labeled $\varepsilon/\varepsilon/\varepsilon$. Once M has finished reading its input, it can jump to its corresponding state in M'. From there, it will be able perform any stack operations that M could have performed by reading additional input characters. M^* accepts $pref(L)$ because (informally), it will accept any string w iff w drives the M part of M^* to some configuration $c = (q, \varepsilon, s)$, where q is a state and s is a stack value, and there is some string w' that could have driven M from (q, w', s) to some other configuration $(p, \varepsilon, \varepsilon)$ where p is an accepting state.

We can also show this by construction of a context-free grammar: If L is CF, then there exists some cfg G that generates it. Let S be the start symbol of G. If $\varepsilon \notin L$, then convert G to G', a grammar that also generates L but is in Chomsky Normal Form. If $\varepsilon \in L$, then convert G to G', a Chomsky Normal Form grammar that generates $L - \{\varepsilon\}$.

The basic idea behind the construction: Every parse tree generated by a grammar in Chomsky Normal Form is a binary tree. It looks like:



and so forth. To generate the prefixes of all the strings in L, we need to be able to pick a point in the tree and then, for every node to the right of that point, generate ε whenever the original grammar would have generated some terminal symbol. So, for example, maybe the A subtree will be complete but somewhere in the B subtree, we'll stop generating terminal symbols. Note that if the B subtree generates any terminals, then the A subtree must be complete. Also, note that if the Z subtree is incomplete then the Q subtree must generate no terminals. But maybe the A subtree will be incomplete, in which case the B subtree must generate no terminal symbols. To make this work, for every nonterminal A in G' we will introduce two nonterminals: A_C and A_E. The interpretations we will assign to the three A's are:

- A_C will generate only complete strings (i.e., those that could be generated by A in G').
- A_E will generate no terminal symbols. In other words, whenever G' would have generated a terminal symbol, A_E will generate ε.
- A will generate all initial substrings of any string generated by A in G'. In other words, it may start by generating terminals, but it may quit at any point and switch to ε.

Construct G_P to generate $pref(L)$: G_P is initially empty. Rules will be added as follows:

- If $\varepsilon \in L$, then create a new start symbol S' and add the rules:
  - $S' \rightarrow \varepsilon$ (This may be necessary since it may be the only way to generate ε, which
  - $S' \rightarrow S$ cannot be generated by G'.)
- For each rule $X \rightarrow A\ B$ in G', add the following rules to describe how to build some initial substring of a string that could be generated by X in G':
  - $X \rightarrow A_C\ B$ (corresponding to building a complete A subtree and then a possibly incomplete B one)
  - $X \rightarrow A\ B_E$ (corresponding to building a possibly incomplete subtree for A followed by an empty one for B)

- For each rule $X \rightarrow A\ B$ in G', add the following rule to describe how to build a complete tree rooted at X:
  - $X_C \rightarrow A_C\ B_C$
- For each rule $X \rightarrow A\ B$ in G', add the following rule to describe how to build an empty tree rooted at X:
  - $X_E \rightarrow A_E\ B_E$
- For each rule $X \rightarrow a$ in G', add the following rules to generate either terminal symbols or ε:
  - $X \rightarrow a$ (since X can be either complete or incomplete)
  - $X \rightarrow \varepsilon$
  - $X_C \rightarrow a$ (since X_C must be complete)
  - $X_E \rightarrow \varepsilon$ (since X_E must be empty)

d) $middle(L) = \{x: \exists y, z \in \Sigma^* (yxz \in L)\}$.

Closed. The proof is by a construction similar to the one given for *init* except that we build two extra copies of M, both of which mimic all of M's transitions except they read no input. From each state in copy one, there is a transition labeled ε/ε/ε to the corresponding state in M, and from each state in M there is a transition labeled ε/ε/ε to the corresponding state in the second copy. The start state of M^* is the start state of copy 1. So M^* begins in the first copy, performing, without actually reading any input, whatever stack operations M could have performed while reading some initial input string y. At any point, it can guess that it's skipped over all the characters in y. So it jumps to M and reads x. At any point, it can guess that it's read all of x. Then it jumps to the second copy, in which it can do whatever stack operations M would have done on reading z. If it guesses to do that before it actually reads all of x, the path will fail to accept since it will not be possible to read the rest of the input.

e) Letter substitution.

Closed. If L is a context-free language, then it is generated by some context-free grammar G. From G we construct a new grammar G' that generates $letsub(L)$, for any letter substitution function *letsub* defined with respect to a substitution function *sub*.

1. Initially, let $G' = G$.
2. For each rule in G that has any nonterminals on its right-hand side do:
   2.1. Replace each instance of a nonterminal symbol c by $sub(c)$.

$letsub(L)$ must be context-free because it is generated by a context-free grammar.

f) $shuffle(L) = \{w : \exists x \in L\ (w \text{ is some permutation of } x)\}$.

No. Let $L = (\texttt{abc})^*$. Then $shuffle(L) = \{w \in \{\texttt{a}, \texttt{b}, \texttt{c}\}^* : \#_{\texttt{a}}(w) = \#_{\texttt{b}}(w) = \#_{\texttt{c}}(w)\}$, which is not context-free.

g) $copyreverse(L) = \{w : \exists x \in L\ (w = xx^R)\}$.

No. Let $L = WW^R$. Then $copyandreverse(L) = \{w \in \{\texttt{a}, \texttt{b}\}^* : w = xx^Rxx^R\}$, which is not context-free. Prove by pumping.

14) Let $alt(L) = \{x: \exists y,n\ (y \in L\ ,\ |y| = n,\ n > 0,\ y = a_1 \ldots a_n, \forall i \le n\ (a_i \in \Sigma)$, and $x = a_1 a_3 a_5 \ldots a_k$, where $k =$ (if n is even then n-1 else n))$\}$.

a) Consider $L = \mathtt{a}^n\mathtt{b}^n$. Clearly describe $L_1 = alt(L)$.

We must take each string in L and generate a new string that contains every other character of the original string. We'll get one result when the length of the original string was even and one when it was odd. So:

$alt(L) =$ $\mathtt{a}^n\mathtt{b}^n$ /* each original string where n was even produces $\mathtt{a}^{n/2}\mathtt{b}^{n/2}$

$\cup$

$\mathtt{a}^{i+1}\mathtt{b}^i,\ i \ge 0$ /* each original string where n was odd produces $\mathtt{a}^{\lfloor n/2 \rfloor}\mathtt{b}^{\lfloor n/2 \rfloor}$

b) Are the context free languages closed under the function *alt*? Prove your answer.

Closed. We can prove this by construction. If L is context free, then there exists a PDA M that accepts L. We construct a new PDA M* as follows:

1. Initially, let M* equal M.
2. For each state s of M, make a duplicate s*. The basic idea is that we will use the second set of states so that we can distinguish between even and odd characters.
3. For each transition $P = (s, i, w), (t, x)$ in M do:
   i. Create the transition $(s, i, w), (t^*, x)$.
   ii. Create the transition $(s^*, \varepsilon, w), (t, x)$.
   iii. Delete P.

We've constructed M* to mimic M, but with every odd character moving the machine from an original to a duplicate and every even character moving it from a duplicate back to the original. But then we made one change. Every transition from a duplicate back to an original (which should have corresponded to an even numbered character) is labeled ε instead of with the character. So M* makes the move without the character. Thus it accepts strings that would have been in the original language L except that every even numbered character has been removed. Note that whatever stack operations should have been performed are still performed.

15) Let $L_1 = \{\mathtt{a}^n\mathtt{b}^m : n \ge m\}$. Let $R_1 = \{(\mathtt{a} \cup \mathtt{b})^* :$ there is an odd number of a's and an even number of b's$\}$. Use the construction described in the book to build a PDA that accepts $L_1 \cap R_1$.

We start with M_1 and M_2, then build M_3:

M_1, which accepts $L_1 = (\{1, 2\}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}\}, \Delta, 1, \{2\}), \Delta =$
((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 $R_1 = (\{1, 2, 3, 4\}, \{\mathtt{a}, \mathtt{b}\}, \delta, 1, \{2\}), \delta =$
(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 $L_1 \cap R_1 = (\{(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2,), (2, 3), (2, 4)\}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}\}, \Delta, (1,1), \{(2,2)\}), \Delta =$

(((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))

16) Let T be a set of languages defined as follows:

$T = \{L : L$ is a context-free language over the alphabet $\{a, b, c\}$ and, if $x \in L$, then $|x| \equiv_3 0\}$.

Let P be the following function on languages:

$P(L) = \{w : \exists x \in \{a, b, c\}$ and $\exists y \in L$ and $y = xw\}$.

Is the set T closed under P? Prove your answer.

No. Let $L_1 = \{a^k : k \equiv_3 0\}$. Then $P(L_1) = \{a^k : k+1 \equiv_3 0\}$. So $aa \in P(L_1)$. So $P(L_1) \notin T$.

17) Show that the following languages are deterministic context-free:
   a) $\{w : w \in \{a, b\}^*$ and each prefix of w has at least as many a's as b's$\}$.
   b) $\{a^n b^n, n \geq 0\} \cup \{a^n c^n, n \geq 0\}$.

18) Show that $L = \{a^n b^n, n \geq 0\} \cup \{a^n b^{2n}, n \geq 0\}$ is not deterministic context-free.

19) Are the deterministic context-free languages closed under reverse? Prove your answer.

No. Let $L = \{1a^n b^{2n} \cup 2a^n b^n\}$. It is easy to build a deterministic PDA that accepts L. After reading the first input symbol, it knows which sublanguage an input string is in. But no deterministic PDA exists to accept L^R because it isn't possible to know, until the end, whether each a matches one b or two.

20) Prove that each of the following languages L is not context-free. (Hint: use Ogden's Lemma.)
   a) $\{a^i b^j c^k : i \geq 0, j \geq 0, k \geq 0$, and $i \neq j \neq k\}$.

Let k be the constant from Ogden's Lemma. Let $w = a^k b^{k+k!} c^{k+2k!}$. Mark all and only the a's in w as distinguished. If either v or y contains more than one distinct symbol, pump in once. The resulting string will not be in L because it will have letters out of order. Call the a's region 1, the b's region 2, and the c's region 3. At least one of v and y must be in region 1. So there are three possibilities:

(1, 1): let p be the $|vy|$. Then $p \leq k$. So p divides $k!$. Let $q = k!/p$. Pump in q-1 times. The resulting string is $a^{k+qp} b^{k+k!} c^{k+2k!} = a^{k+k!} b^{k+k!} c^{k+2k!}$, which is not in L.
(1, 2): similarly, only let p be the $|v|$ let $q = 2k!/p$. The resulting string is $a^{k+2qp} b^{k+k!} c^{k+2k!} = a^{k+2k!} b^{k+k!} c^{k+2k!}$, which is not in L.
(1, 3): in this case, $|vxy|$ would be greater than k, so it need not be considered.

   b) $\{a^i b^j c^k d^n : i \geq 0, j \geq 0, k \geq 0, n \geq 0$, and $(i = 0$ or $j = k = n\}$.

Let k be the constant from Ogden's Lemma. Let $w = a^k b^k c^k d^k$. Mark all and only the b's in w as distinguished. If either v or y contains more than one distinct symbol, pump in once. The resulting string will not be in L because it will have letters out of order. At least one of v and y must be in the b region.

Pump in once. The resulting string will still have a non-zero number of a's. Its number of b's will have increased and at most one of the c's and d's can have increased. So there are no longer equal numbers of b's, c's, and d's. So the resulting string is not in L.

21) Let $\Psi(L)$ be as defined in Section 13.7, in our discussion of Parikh's Theorem. For each of the following languages L, first state what $\Psi(L)$ is. Then give a regular language that is letter-equivalent to L.

a) Bal = $\{w \in \{), (\}^* : \text{the parentheses are balanced}\}$.

$\Psi(L) = \{(i, i) : 0 \le i\}$. Using [] as metacharacters and () as elements of Σ, the language [()]* is regular and is letter-equivalent to L.

b) Pal = $\{w \in \{a, b\}^* : w \text{ is a palindrome}\}$.

Pal contains both even and odd length palindromes. If even length, then there must be an even number of a's and an even number of b's. If odd length, then one of the two letters has an even count and the other has an odd count. So $\Psi(L) = \{(2i, 2j) : 0 \le i \text{ and } 0 \le j\} \cup \{(2i+1, 2j) : 0 \le i \text{ and } 0 \le j\} \cup \{(2i, 2j+1) : 0 \le i \text{ and } 0 \le j\}$. The regular language $(aa)^*(bb)^*(a \cup b \cup \varepsilon)$ is letter-equivalent to L.

c) $\{x^R\#y : x, y \in \{0, 1\}^* \text{ and } x \text{ is a substring of } y\}$.

Let Σ be $\{0, 1, \#\}$. Then $\Psi(L) = \{(i, j, 1) : 0 \le i \text{ and } 0 \le j\}$. The regular language $(0 \cup 1)^*\#$ is letter equivalent to L.

22) For each of the following claims, state whether it is *True* or *False*. Prove your answer.

a) If L_1 and L_2 are two context-free languages, $L_1 - L_2$ must also be context-free.

False. Let $L_1 = (a \cup b \cup c)^*$. L_1 is regular and thus context-free. Let L_2 be $\neg\{a^n b^n c^n : n \le 0\}$. We showed in the book that L_2 is context-free. But $\neg L_2$ is not context-free.

b) If L_1 and L_2 are two context-free languages and $L_1 = L_2 L_3$, then L_3 must also be context-free.

False. Let $L_1 = aaa^*$, which is regular and thus context-free. Let $L_2 = a^*$, which is also regular and thus context-free. Let L_3 be $\{a^p$, where p is prime$\}$, which is not regular.

c) If L is context free and R is regular, R - L must be context-free.

False. R - L need not be context free. If we let $R = \Sigma^*$, then R - L is exactly the complement of L. So, if R - L is necessarily context-free, then $\neg L$ must also be context-free. But this is not guaranteed to be so since the context-free languages are not closed under complement. As a concrete example that shows that the claim is false, let $R = a^*b^*c^*$ and $L = \{a^i b^j c^k$, where $i \ne j$ or $j \ne k\}$. R - $L = A^n B^n C^n = a^n b^n c^n$, which is not context free.

d) If L_1 and L_2 are context-free languages and $L_1 \subseteq L \subseteq L_2$, then L must be context-free.

False. Let $L_1 = \varnothing$. Let $L_2 = (a \cup b \cup c)^*$. Let $L = \{a^n b^n c^n , n \ge 0\}$, which is not context-free.

e) If L_1 is a context-free language and $L_2 \subseteq L_1$, then L_2 must be context-free.

False. Let $L_1 = a^*$. Let $L_2 = \{a^p$, where p is prime$\}$. L_2 is not context-free.

f) If L_1 is a context-free language and $L_2 \subseteq L_1$, it is possible that L_2 is regular.

True. Let L_1 and L_2 be a^*.

g) A context-free grammar in Chomsky normal form is always unambiguous.

False. Any context-free grammar can be converted to Chomsky normal form. But there are inherently ambiguous context-free languages.

14 Decision Procedures for Context-Free Languages

1) Give a decision procedure to answer each of the following questions:

   a) Given a regular expression α and a PDA M, is the language accepted by M a subset of the language generated by α?

   Observe that this is true iff $L(M) \cap L(\alpha) = \varnothing$. So the following procedure answers the question:

   1. From α, build a PDA M^* so that $L(M^*) = L(\alpha)$.
   2. From M and M^*, build a PDA M^{**} that accepts $L(M) \cap L(M^*\alpha)$
   3. If $L(M^{**})$ is empty, return *True*, else return *False*.

   b) Given a context-free grammar G and two strings s_1 and s_2, does G generate s_1s_2?

   1. Convert G to Chomsky Normal Form.
   2. Try all derivations in G of length up to $2|s_1s_2|$. If any of them generates s_1s_2, return *True*, else return *False*.

   c) Given a context-free grammar G, does G generate at least three strings?

   d) Given a context-free grammar G, does G generate any even length strings?

   1. Use *CFGtoPDAtopdown*(G) to build a PDA P that accepts $L(G)$.
   2. Build an FSM E that accepts all even length strings over the alphabet Σ_G.
   3. Use *intersectPDAandFSM*(P, E) to build a PDA P^* that accepts $L(G) \cap L(E)$.
   4. Return *decideCFLempty*(P^*).

   e) Given a regular grammar G, is $L(G)$ context-free?

   1. Return *True* (since every regular language is context-free).

15 Parsing

1) Consider the following grammar that we presented in Example 15.9:

   $S \rightarrow AB\$ \mid AC\$$
   $A \rightarrow \texttt{a}A \mid \texttt{a}$
   $B \rightarrow \texttt{b}B \mid \texttt{b}$
   $C \rightarrow \texttt{c}$

   Show an equivalent grammar that is LL(1) and prove that it is.

   $S \rightarrow AX$
   $A \rightarrow \texttt{a}Y$
   $Y \rightarrow A \mid \varepsilon$
   $X \rightarrow B\$ \mid C\$$
   $B \rightarrow \texttt{b}Z$
   $Z \rightarrow B \mid \varepsilon$
   $C \rightarrow \texttt{c}$

2) Assume the grammar:

   $S \rightarrow NP\ VP$
   $NP \rightarrow ProperNoun$
   $NP \rightarrow Det\ N$
   $VP \rightarrow V\ NP$
   $VP \rightarrow VP\ PP$
   $PP \rightarrow Prep\ NP$

   Assume that `Jen` and `Bill` have been tagged *ProperNoun*, `saw` has been tagged *V*, `through` has been tagged *Prep*, `the` has been tagged *Det*, and `window` has been tagged *N*. Trace the execution of *Earleyparse* on the input sentence `Jen saw Bill through the window.`

3) Trace the execution of *Earleyparse* given the string and grammar of Example 15.5.

4) Trace the execution of a CKY parser on the input string `id + id * id`, given the unambiguous arithmetic expression grammar shown in Example 11.19, by:
   a) Converting the grammar to Chomsky normal form.
   b) Showing the steps of the parser.

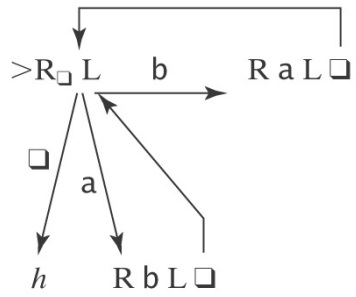
16 Summary and References

Part IV: Turing Machines and Undecidability

17 Turing Machines

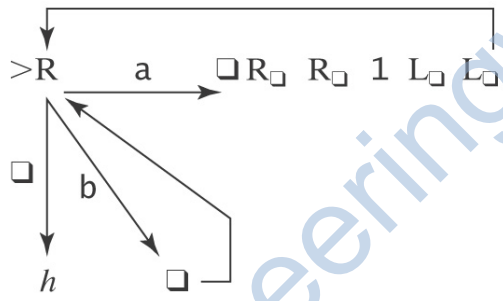
1) Give a short English description of what each of these Turing machines does:

a) $\Sigma_M = \{a, b\}$. $M =$



Shift the input string one character to the right and replace each b with an a and each a with a b.

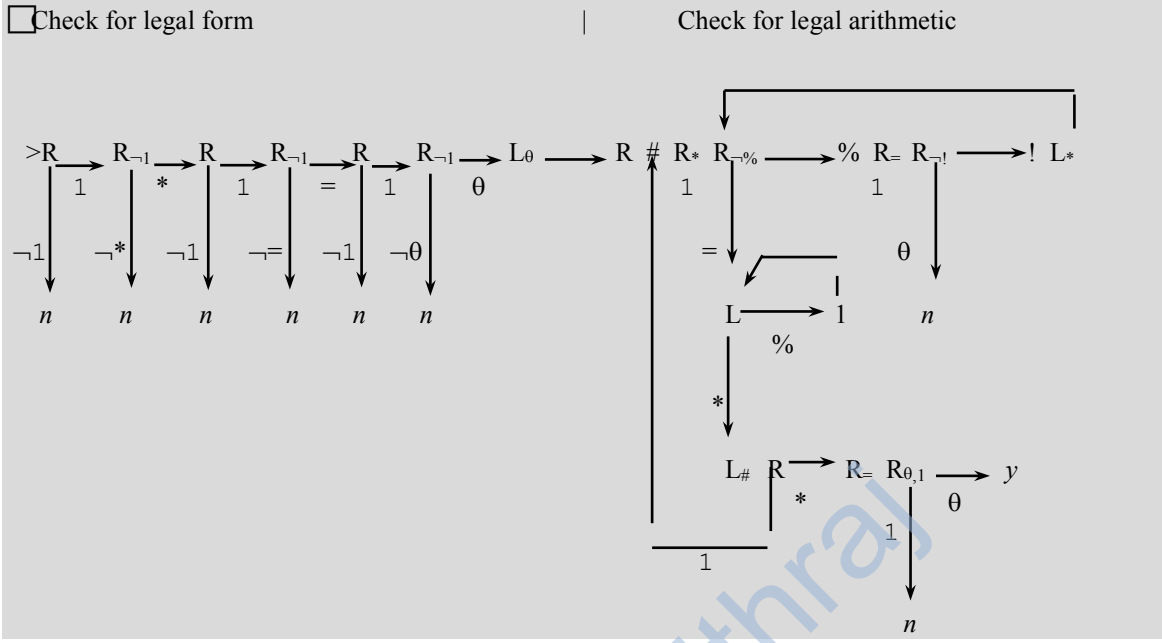
b) $\Sigma_M = \{a, b\}$. $M =$



Erase the input string and replace it with a count, in unary, of the number of a's in the original string.

2) Construct a standard, one-tape Turing machine M to decide each of the following languages L. You may find it useful to define subroutines. Describe M in the macro language described in Section 17.1.5.

a) $\{x * y = z : x, y, z \in 1^+$ and, when x, y, and z are viewed as unary numbers, $xy = z\}$. For example, the string 1111*11=11111111 $\in L$.



b) $\{a^i b^j c^i d^j, i, j \geq 0\}$.

c) $\{w \in \{a, b, c, d\}^*: \#_b(w) \geq \#_c(w) \geq \#_d(w) \geq 0\}$.

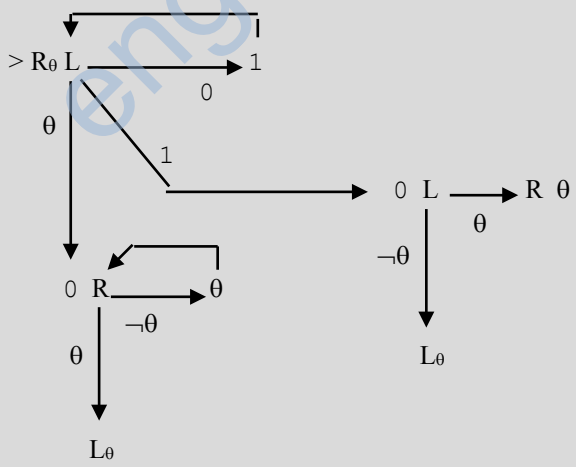
3) Construct a standard 1-tape Turing machine M to compute each of the following functions:

a) The function sub_3, which is defined as follows:

$sub_3(n) =$ n-3 if $n > 2$
0 if $n \leq 2$.

Specifically, compute sub_3 of a natural number represented in binary. For example, on input 10111, M should output 10100. On input 11101, M should output 11010. (Hint: you may want to define a subroutine.)

We first define a subroutine that we will call S to subtract a single 1:

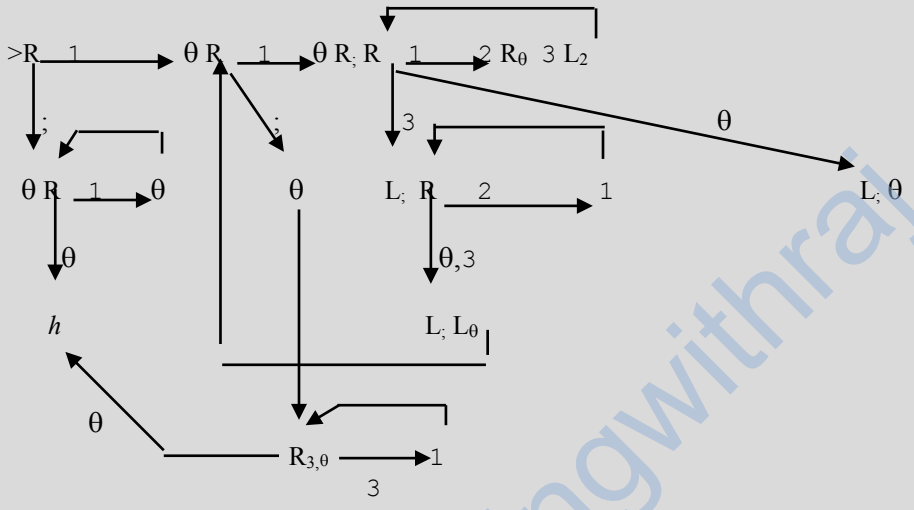


Now we can define M as: $> S S S$

b) Addition of two binary natural numbers (as described in Example 17.13). Specifically, given the input string <x>;<y>, where <x> is the binary encoding of a natural number x and <y> is the binary encoding of

a natural number y, M should output <z>, where z is the binary encoding of $x + y$. For example, on input `101;11`, M should output `1000`.

c) Multiplication of two unary numbers. Specifically, given the input string <x>;<y>, where <x> is the unary encoding of a natural number x and <y> is the unary encoding of a natural number y, M should output <z>, where z is the unary encoding of xy. For example, on input `111;1111`, M should output `111111111111`.



This machine first erases the first `1` in x. Then it uses each of the others to create a new copy of y. Each time it uses a `1` in x, it writes over it with a blank. Once all the ones in x have created their copies of y, the ; is erased.

d) The proper subtraction function *monus*, which is defined as follows:

$$monus(n, m) = \begin{cases} n - m & \text{if } n > m \\ 0 & \text{if } n \leq m \end{cases}$$

Specifically, compute *monus* of two natural numbers represented in binary. For example, on input `101;11`, M should output `10`. On input `11;101`, M should output `0`.

4) Define a Turing Machine M that computes the function f: {a, b}* → N, where:

$f(x)$ = the unary encoding of $max(\#_a(x), \#_b(x))$.

For example, on input `aaaabb`, M should output `1111`. M may use more than one tape. It is not necessary to write the exact transition function for M. Describe it in clear English.

We can use 3 tapes:
Tape 1: input
Tape 2: write `1` for every `a` in the input
Tape 3: write `1` for every `b` in the input

- Step 1: Move left to right along tape 1. If the character under the read head is `a`, write `1` on tape 2 and move one square to the right on tapes 1 and 2. If the character under the read is `b`, write `1` on tape 3 and move one square to the right on tapes 1 and 3. When tape 1 encounters a blank, go to step 2.
- Step 2: Move all three read heads all the way to the left. Scan the tapes left to right. At each step, if there is `1` on either tape 2 or 3 (or both), write `1` on tape 1. Move right on all tapes except that as soon

as either tape 2 or tape 3 (but not both) gets to a blank, stop moving right on that tape and continue this process with just the other one. As soon as the other of them (tape 2 or 3) also hits a blank, go to step 3. At this point, tape 1 contains the correct number of 1's, plus perhaps extra a's and b's that still need to be erased.

- Step 3: Scan left to right along tape 1 as long as there are a's or b's. Rewrite each as a blank. As soon as a blank is read, stop.

5) Construct a Turing machine *M* that converts binary numbers to their unary representations. So, specifically, on input <*w*>, where *w* is the binary encoding of a natural number *n*, *M* will output 1^n. (Hint: use more than one tape.)

M will use three tapes. It will begin by copying its input to tape 2, where it will stay, unchanged. Tape 1 will hold the answer by the end. Tape 3 will hold a working string defined below. *M* will initialize itself by copying its input to tape 2 and writing 1 on tape 3. Then it will begin scanning tape 2, starting at the rightmost symbol, which we'll call symbol 0. As *M* computes, tape 3 will contain 1^i if *M* is currently processing the i^{th} symbol (from the right, starting numbering at 0). Assume that *M* has access to a subroutine *double* that will duplicate whatever string is on tape 3. So if that string is *s*, it will become *ss*. After initialization, *M* operates as follows:

For each symbol *c* on tape 2, do:

1. If *c* = 1, then append a copy of the nonblank region of tape 3 to the end of the nonblank region of tape 1. (If this is the first append operation, just write the copy on the tape where the read/write head is.)
2. Call *double*.
3. Move the read head on tape 2 one square to the left.
4. If the square under the read/write head on tape 2 is θ, halt. The answer will be on tape 1.

6) Let *M* be a three-tape Turing machine with Σ = {a, b, c} and Γ = {a, b, c, θ, 1, 2}. We want to build an equivalent one-tape Turing machine *M′* using the technique described in Section 17.3.1 How many symbols must there be in Γ′?

7) In Example 13.2, we showed that the language $L = \{a^{n^2}, n \geq 0\}$ is not context-free. Show that it is in D by describing, in clear English, a Turing machine that decides it. (Hint: use more than one tape.)

8) In Example 17.9, we showed a Turing machine that decides the language WcW. If we remove the middle marker c, we get the language WW. Construct a Turing machine *M* that decides WW. You may exploit nondeterminism and/or multiple tapes. It is not necessary to write the exact transition function for *M*. Describe it in clear English.

M's first job is to find the middle of the string. It will shift the first half of the string one square to the left, then it will deposit # between the two halves. So *M* moves the first character one square to the left. Then it nondeterministically decides whether it has moved half the string. If it decides it has not, it moves one square to the right and shifts it one square to the left, and continues doing that until it decides it's halfway through. At that point, it writes # between the two halves. Now *M* must compare the two halves. It does this by bouncing back and forth, marking off the first character and the first character after #. Then the second, and so forth. If it comes out even with all characters matching, it accepts. If either there is a mismatch or one string is longer than the other, that path rejects. If the # is in the middle, then *M* operates like the *w*c*w* machine described in the book.

9) In Example 4.9, we described the Boolean satisfiability problem and we sketched a nondeterministic program that solves it using the function *choose*. Now define the language SAT = {<*w*> : *w* is a wff in Boolean logic and *w* is satisfiable}. Describe in clear English the operation of a nondeterministic (and possibly *n*-tape) Turing machine that decides SAT.

We will assume that a logical predicate symbol will be encoded as a single symbol that is not one of the logical symbols. *M* will use two tapes and operate as follows:

1. Move right along the input tape (tape 1) looking for a logical predicate symbol or a θ. If a θ is found, go to step 6.
2. Remember the predicate symbol in the variable *p*.
3. Nondeterministically choose whether to replace this symbol by *T* or *F*.
4. Write whichever of those values was chosen.
5. Move right looking for any additional instances of *p* or a θ. Whenever an instance of *p* is found, replace it by the chosen value, *T* or *F*. When a θ is found, move the read/write head left to the first blank and go back to step 1.
6. At this point, all predicate symbols have been replaced by *T* or *F*, so it remains to evaluate the expression. Because parentheses are allowed, *M* will use its second tape as a stack. *M* will simulate the execution of a shift-reduce parser, whose input is on tape one and whose stack is on tape 2. Whenever *M* simulates a reduction that involves a Boolean operator, it will place onto its stack the result of applying that operator to its operands.
7. If *M* succeeds in parsing its entire input and writing a single value, *T* or *F*, on tape 2, then, if it wrote *T*, it will accept. If it wrote *F*, it will reject. If it doesn't succeed in the parse, it will reject (because the input was ill-formed and thus not in SAT).

10) Prove Theorem 17.3, which tells us that, if a nondeterministic Turing machine *M* computes a function *f*, then there exists a deterministic TM *M′* that computes *f*.

11) Prove rigorously that the set of regular languages is a *proper* subset of D.

Lemma: The set of regular languages is a subset of D:

Proof of Lemma: Given any regular language *L*, there exists a DFSM *F* that accepts *L*. From *F*, we construct a TM *M* that decides *L*: *M* contains the same set of states as *F*, plus three new states, *s′*, *y* and *n*. We make *s′* the start state of *M*. If *s* is the start state of *F*, we create in *M* the transition ((*s′*, θ), (*s*, θ, →)), which moves right one square and is then pointing at the first nonblank character on the tape. For every transition (*q*, *a*, *r*) in *F*, we create in *M* the transition ((*q*, *a*), (*r*, *a*, →)). Thus *M* reads its input characters one at a time, just as *F* does. For every accepting state *q* in *F*, we create in *M* the transition ((*q*, θ), (*y*, θ, →). For every nonaccepting state *q* in D, we create in *M* the transition ((*q*, θ) (*n*, θ, →)). So, as soon as *M* has read all of its input characters, it goes to *y* iff *F* would have accepted and it goes to *n* otherwise. *M* always halts and it accepts exactly the same strings that *F* does. So *M* decides *L*. Since there is a TM that decides *L*, *L* is in D.

We now prove that the set of regular languages is a *proper* subset of the set of recursive languages.

$L = \{a^n b^n c^n : n \leq 0\}$ is in D. We defined a TM to decide it. But *L* is not regular. (Proof: use the pumping theorem.) There exists at least one element of D that is not regular. So if the set of regular languages is a subset of D it is a proper subset. But it is a subset (as shown above). So it is a proper subset.

12) In this question, we explore the equivalence between function computation and language recognition as performed by Turing machines. For simplicity, we will consider only functions from the nonnegative integers to the nonnegative integers (both encoded in binary). But the ideas of these questions apply to any computable function. We'll start with the following definition:

- Define the *graph* of a function *f* to be the set of all strings of the form [*x*, *f*(*x*)], where *x* is the binary encoding of a nonnegative integer, and *f*(*x*) is the binary encoding of the result of applying *f* to *x*.

For example, the graph of the function *succ* is the set {[0, 1], [1, 10], [10, 11], …}.

a) Describe in clear English an algorithm that, given a Turing machine M that computes f, constructs a Turing machine M' that decides the language L that contains exactly the graph of f.

Let M be the TM that that computes f. We construct a two-tape TM, M' that accepts the graph of f as a language: M' takes as input a string of the form $[x,y]$. Otherwise it halts and rejects. M' copies x onto its second tape. It then runs M on that tape. When M halts, M' compares the result on tape 2 with y on tape 1. If they are the same, it halts and accepts. Otherwise, it halts and rejects.

b) Describe in clear English an algorithm that, given a Turing machine M that decides the language L that contains the graph of some function f, constructs a Turing machine M' that computes f.

There exists a TM M that decides L. We construct a two-tape M' to compute f as follows: On input x, write, on tape 1, $[x, 1]$. Copy that onto tape 2. Run M on tape 2. If it accepts, erase [x and] from tape 1, leaving 1 and halt. That's the answer. If it rejects, increment the second number on tape 1. Copy tape 1 to tape 2, and again run M. Continue until M eventually halts and accepts (which it must do since f is a function and so must return some value for any input) and report the answer on tape 1.

c) A function is said to be partial if it may be undefined for some arguments. If we extend the ideas of this exercise to partial functions, then we do not require that the Turing machine that computes f halts if it is given some input x for which $f(x)$ is undefined. Then L (the graph language for f), will contain entries of the form $[x, f(x)]$ for only those values of x for which f is defined. In that case, it may not be possible to decide L, but it will be possible to semidecide it. Do your constructions for parts (a) and (b) work if the function f is partial? If not, explain how you could modify them so they will work correctly. By "work", we mean:

- For part (a): given a Turing machine that computes $f(x)$ for all values on which f is defined, build a Turing machine that semidecides the language L that contains exactly the graph of f;
- For part (b): given a Turing machine that semidecides the graph language of f (and thus accepts all strings of the form$[x, f(x)]$ when $f(x)$ is defined), build a Turing machine that computes f.

For (a): We need to build a TM M' that halts on any $[x,y]$ pair such that f is defined on x and $f(x) = y$. Otherwise M' must loop. To make this happen, we have to make one change to the solution given above: M' copies x onto its second tape. It then runs M on that tape. If M halts, M' compares the result on tape 2 with y on tape 1. If they are the same, it halts and accepts. Otherwise, it loops. M' will loop if either f is not defined on x or $f(x) \neq y$. Otherwise (i.e., when $f(x) = y$) , it will halt.

For (b), we must also make one change. Instead of checking possible values for y sequentially, we must check them in dovetailed mode. If $f(x)$ is defined, M will accept some string of the form $[x,y]$ and so M' will eventually halt with y on its tape. If $f(x)$ is undefined then M will accept no string of the form $[x,y]$. So no branch of M' will halt.

13) What is the minimum number of tapes required to implement a universal Turing machine?

One. It can be implemented with three tapes as described in the book. But for every three-tape Turing machine, there is an equivalent one-tape one.

14) Encode the following Turing Machine as an input to the universal Turing machine:

$M = (K, \Sigma, \Gamma, \delta, q_0, \{h\})$, where:
$K = \{q_0, q_1, h\}$,
$\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
$\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \theta\}$, and
δ is given by the following table:

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, b, \rightarrow)$
q_0	b	$(q_1, a, \rightarrow)$
q_0	θ	$(h, \theta, \rightarrow)$
q_0	c	$(q_0, c, \rightarrow)$
q_1	a	$(q_0, c, \rightarrow)$
q_1	b	$(q_0, b, \leftarrow)$
q_1	θ	$(q_0, c, \rightarrow)$
q_1	c	$(q_1, c, \rightarrow)$

We can encode the states and the alphabet as:

q_0	q00
q_1	q01
h	q10
a	a00
b	a01
θ	a10
c	a11

We can then encode δ as:

```
(q00,a00,q01,a01,→),(q00,a01,q01,a00,→),(q00,a10,q10,a10,→),
(q00,a11,q00,a11,→),(q01,a00,q00,a11,→),(q01,a01,q00,a01,←),
(q01,a10,q00,a11,→),(q01,a11,q01,a11,→)
```

18 The Church-Turing Thesis

1) Church's Thesis makes the claim that all reasonable formal models of computation are equivalent. And we showed in, Section 17.4, a construction that proved that a simple accumulator/register machine can be implemented as a Turing machine. By extending that construction, we can show that any computer can be implemented as a Turing machine. So the existence of a decision procedure (stated in any notation that makes the algorithm clear) to answer a question means that the question is decidable by a Turing machine.

   Now suppose that we take an arbitrary question for which a decision procedure exists. If the question can be reformulated as a language, then the language will be in D iff there exists a decision procedure to answer the question. For each of the following problems, your answers should be a precise description of an algorithm. It need not be the description of a Turing Machine:

   a) Let $L = \{\langle M \rangle : M$ is a DFSM that doesn't accept any string containing an odd number of `1`'s$\}$. Show that L is in D.

   If M has n states and M accepts any strings that contain an odd number of `1`'s, then M must accept at least one such string of length $\leq 2n$. Why? If M accepts any strings that contain an odd number of `1`'s, then there are two possibilities: 1) There is a string of length less than n that is accepted by M and that contains an odd number of `1`'s. 2) There is no string of length less than n that is accepted by M and that contains an odd number of `1`'s but there is a longer such string. Call it s. Then M must have accepted s by traversing at least one loop L that contained an odd number of `1`'s (because there are no strings without loops with an odd number of `1`'s). Pump out of s all substrings corresponding to all loops except one instance of L. The resulting string s' must be accepted by M. It must contain an odd number of `1`'s and it must be of length less than $2n$.

   So, an algorithm to decide L is:
   1. Enumerate all strings over Σ of length $\leq 2n$.
   2. Run M on each such string s. If M accepts s and s contains an odd number of `1`'s, reject.
   3. If all strings have been checked and we have not yet rejected, then accept.

   b) Let $L = \{\langle E \rangle : E$ is a regular expression that describes a language that contains at least one string w that contains `111` as a substring$\}$. Show that L is in D.

   An algorithm to decide L:
   1. Construct a DFSM M from E.
   2. Construct S, the set of states of M that are reachable from the start state of M.
   3. Construct T, the set of states of M from which there exists some path to some final state.
   4. For all $q \in S$ do:
      Simulate M on 111 starting in q. If M lands in some state $s \in T$ then halt and accept.
   5. If no path was accepted in the previous step, halt and reject.

   c) Consider the problem of testing whether a DFSM and a regular expression are equivalent. Express this problem as a language and show that it is in D.

   Let $L = \{\langle F \rangle \# R : L(F) = L(R)\}$, where $\langle F \rangle$ is a string that encodes a DFSM and R is a regular expression. (Note that regular expressions are already strings, so we need no special encoding of them.)

   An algorithm to decide L:
   1. From R, construct a FSM M that accepts $L(R)$.
   2. From M, construct M_M, a minimal deterministic FSM equivalent to M.
   3. From F, construct F_M, a minimal deterministic FSM equivalent to F.
   4. If M_M and F_M are equivalent (i.e., they are isomorphic as graphs), accept. Else reject.

2) Consider the language $L = \{w = xy : x, y \in \{\texttt{a}, \texttt{b}\}^*$ and y is identical to x except that each character is duplicated$\}$. For example $\texttt{ababaabbaabb} \in L$.

a) Show that L is not context-free.

If L were context-free, then $L' = L \cap \texttt{a}^*\texttt{b}^*\texttt{a}^*\texttt{b}^*$ would also be context-free. But it isn't, which we can show using the Pumping Theorem. Let w be:

$$\texttt{a}^k\texttt{b}^k\texttt{a}^{2k}\texttt{b}^{2k}$$
$$1 \mid 2 \mid 3 \mid 4$$

If either v or y crosses region boundaries, pump in once; the resulting string will not be in L' because it violates the form constraints. Also note that every string in L has length that is divisible by 3. So, if $|vy|$ is not divisible by 3, pump in once and the resulting string will not be in L'. We now consider the other possibilities. In doing so, we can think of every string in L' as having a first part, which is the first third of the string, and a second part, which is the last two thirds of the string. The second part must contain both twice as many a's and twice as many b's as the first part. Call the string that results from pumping w'.

(1, 1), (2, 2), (1, 2): Set q to 2. For every 3 symbols in vy, 2 symbols from the right end of the first part of w move into the second part of w'. So the second part of w' starts with a b, while the first part starts with an a. Thus w' is not in L and thus also not in L'.
(3, 3), (4, 4), (3, 4): Set q to 2. For every 3 symbols in vy, 1 symbol from the left end of the second part of w moves into the first part of w'. So the second part of w' ends with a b, while the first part ends with an a. Thus w' is not in L and thus also not in L'.
(2, 3): Set q to 2. If $|y| = 2 \cdot |v|$, then the boundary between the first part of w' and its second part still falls at the end of the first b region. But the number of a's in the first part has grown yet the number of a's in the second part hasn't. So w' is not in L'. If $|y| < 2 \cdot |v|$, then the boundary between the first part of w' and its second part shifts and occurs somewhere inside the first b region. So w' is not in L' because its first part starts with an a while its second part starts with a b. Similarly, if If $|y| > 2 \cdot |v|$, then the boundary between the first part of w' and its second part shifts and occurs somewhere inside the second a region. So w' is not in L' because its first part ends with an a while its second part ends with a b.
(1, 3), (2, 4): violate the requirement that $|vxy| \le k$.

b) Show a Post system that generates L.

$P = (\{S, T, \#, \%, \texttt{a}, \texttt{b}\}, \{\texttt{a}, \texttt{b}\}, \{X, Y\}, R, S)$, where $R =$

$S \rightarrow T\#$
$XT\# \rightarrow X\texttt{a}T\#$
$XT\# \rightarrow X\texttt{b}T\#$
$XT\# \rightarrow X\%X\%$
$X\%\texttt{a}Y\% \rightarrow X\texttt{aa}\%Y\%$
$X\%\texttt{b}Y\% \rightarrow X\texttt{bb}\%Y\%$
$X\%\% \rightarrow X$

3) Show a Post system that generates $\texttt{A}^n\texttt{B}^n\texttt{C}^n$.

$P = (\{S, A, B, C, \texttt{a}, \texttt{b}\}, \{\texttt{a}, \texttt{b}, \texttt{c}\}, \{X, Y\}, R, S)$, where $R =$

$S \rightarrow ABC$
$AXBYC \rightarrow A\texttt{a}XB\texttt{b}Y\texttt{c}C$
$AXBYC \rightarrow \texttt{a}X\texttt{b}Y\texttt{c}$

4) Show a Markov algorithm to subtract two unary numbers. For example, on input 111-1, it should halt with the string 11. On input 1-111, it should halt with the string -11.

5) Show a Markov algorithm to decide WW.

6) Consider Conway's Game of Life, as described in Section 18.2.7. Draw an example of a simple Life initial configuration that is an oscillator, meaning that it changes from step to step but it eventually repeats a previous configuration.

19 The Unsolvability of the Halting Problem

1) Consider the language $L = \{<M> : M \text{ accepts at least two strings}\}$.
   a) Describe in clear English a Turing machine M that semidecides L.

   M generates the strings in Σ_M^* in lexicographic order and uses dovetailing to interleave the computation of M on those strings. As soon as two computations accept, M halts and accepts.

   b) Suppose we changed the definition of L just a bit. We now consider:
   $L' = \{<M> : M \text{ accepts } \textit{exactly } 2 \text{ strings}>$.
   Can you tweak the Turing machine you described in part a to semidecide L'?

   No. M could discover that two strings are accepted. But it will never know that there aren't any more.

2) Consider the language $L = \{<M> : M \text{ accepts the binary encodings of the first three prime numbers}\}$.
   a) Describe in clear English a Turing machine M that semidecides L.

   On input <M> do:
   1. Run M on `10`. If it rejects, loop.
   2. If it accepts, run M on `11`. If it rejects, loop.
   3. If it accepts, run M on `101`. If it accepts, accept. Else loop.

   This procedure will halt and accept iff M accepts the binary encodings of the first three prime numbers. If, on any of those inputs, M either fails to halt or halts and rejects, this procedure will fail to halt.

   b) Suppose (contrary to fact, as established by Theorem 19.2) that there were a Turing machine *Oracle* that decided H. Using it, describe in clear English a Turing machine M that decides L.

   On input <M> do:
   1. Invoke *Oracle*(<M, `10`>).
   2. If M would not accept, reject.
   3. Invoke *Oracle*(<M, `11`>).
   4. If M would not accept, reject.
   5. Invoke *Oracle*(<M, `101`>).
   6. If M would accept, accept. Else reject.

20 Decidable and Semidecidable Languages

1) Show that the set D (the decidable languages) is closed under:
   a) Union
   b) Concatenation
   c) Kleene star
   d) Reverse
   e) Intersection

All of these can be done by construction using deciding TMs. (Note that there's no way to do it with grammars, since the existence of an unrestricted grammar that generates some language L does not tell us anything about whether L is in D or not.)

a) Union is straightforward. Given a TM M_1 that decides L_1 and a TM M_2 that decides L_2, we build a TM M_3 to decide $L_1 \cup L_2$ as follows: Initially, let M_3 contain all the states and transitions of both M_1 and M_2. Create a new start state S and add transitions from it to the start states of M_1 and M_2 so that each of them begins in its start state with its read/write head positioned just to the left of the input. The accepting states of M_3 are all the accepting states of M_1 plus all the accepting states of M_2.

b) is a bit tricky. Here it is: If L_1 and L_2 are both in D, then there exist TMs M_1 and M_2 that decide them. From M_1 and M_2, we construct M_3 that decides L_1L_2. Since there is a TM that decides L_3, it is in D.

The tricky part is doing the construction. When we did this for FSMs, we could simply glue the accepting states of M_1 to the start state of M_2 with ε transitions. But that doesn't work now. Consider a machine that enters the state y when it scans off the edge of the input and finds a blank. If we're trying to build M_3 to accept L_1L_2, then there won't be a blank at the end of the first part. But we can't simply assume that that's how M_1 decides it's done. It could finish some other way.

So we need to build M_3 so that it works as follows: M_3 will use three tapes. Given some string w on tape 1, M_3 first nondeterministically guesses the location of the boundary between the first segment (a string from L_1) and the second segment (a string from L_2). It copies the first segment onto tape 2 and the second segment onto tape 3. It then simulates M_1 on tape 2. If M_1 accepts, it simulates M_2 on tape 3. If M_2 accepts, it accepts. If either M_1 or M_2 rejects, that path rejects.

There is a finite number of ways to carve the input string w into two segments. So there is a finite number of branches. Each branch must halt since M_1 and M_2 are deciding machines. So eventually all branches will halt. If at least one accepts, M_3 will accept. Otherwise it will reject.

2) Show that the set SD (the semidecidable languages) is closed under:
   a) Union
   b) Concatenation
   c) Kleene star
   d) Reverse
   e) Intersection

3) Let $L_1, L_2, \ldots, L_k$ be a collection of languages over some alphabet Σ such that:
   - For all $i \neq j$, $L_i \cap L_j = \varnothing$.
   - $L_1 \cup L_2 \cup \ldots \cup L_k = \Sigma^*$.
   - $\forall i$ (L_i is in SD).

   Prove that each of the languages L_1 through L_k is in D.

$\forall i$ ($\neg L_i = L_1 \cup L_2 \cup \ldots \cup L_{i-1} \cup L_{i+1} \cup \ldots \cup L_k$).
Each of these L_j's is in SD, so the union of all of them is in SD. Since L_i is in SD and so is its complement, it is in D.

4) If L_1 and L_3 are in D and $L_1 \subseteq L_2 \subseteq L_3$, what can we say about whether L_2 is in D?

L_2 may or may not be in D. Let L_1 be $\varnothing$ and let L_3 be Σ. Both of them are in D. Suppose L_2 is H. Then it is not in D. But now suppose that L_2 is {a}. Then it is in D.

5) Let L_1 and L_2 be any two decidable languages. State and prove your answer to each of the following questions:

a) Is it necessarily true that L_1 - L_2 is decidable?

Yes. The decidable languages are closed under complement and intersection, so they are closed under difference.

b) Is it possible that $L_1 \cup L_2$ is regular?

Yes. Every regular language is decidable. So Let L_1 and L_2 be {a}. $L_1 \cup L_2$ = {a}, and so is regular.

6) Let L_1 and L_2 be any two undecidable languages. State and prove your answer to each of the following questions:

a) Is it possible that L_1 - L_2 is regular?

Yes. Let $L_1 = L_2$. Then L_1 - $L_2 = \varnothing$, which is regular.

b) Is it possible that $L_1 \cup L_2$ is in D?

Yes. H $\cup$ ¬H = {<M, w>}.

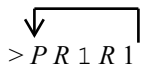
7) Let M be a Turing machine that lexicographically enumerates the language L. Prove that there exists a Turing machine M' that decides L^R.

Since L is lexicographically enumerated by M, it is decidable. The decidable languages are closed under reverse. So L^R is decidable. Thus there is some Turing machine M' that decides it.

8) Construct a standard one-tape Turing machine M to enumerate the language:

{w : w is the binary encoding of a positive integer that is divisible by 3}.

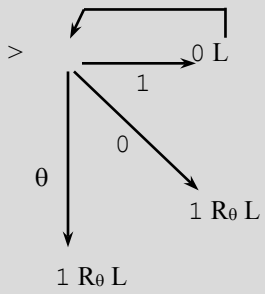
Assume that M starts with its tape equal to $\underline{\theta}$. Also assume the existence of the printing subroutine P, defined in Section 20.5.1. As an example of how to use P, consider the following machine, which enumerates L', where L' = {w : w is the unary encoding of an even number}:



You may find it useful to define other subroutines as well.

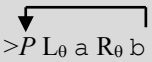
Define the subroutine *A* (Add1) as follows:

Input: θ w_1 w_2 w_3 …$\underline{w_n}$ θ (encoding some integer *k*)
Output: θ w_1 w_2 w_3 …$\underline{w_m}$ θ (encoding *k*+1)



The enumerating machine *M* is now:

9) Construct a standard one-tape Turing machine *M* to enumerate the language A^nB^n. Assume that *M* starts with its tape equal to $\underline{\theta}$. Also assume the existence of the printing subroutine *P*, defined in Section 20.5.1.



10) If *w* is an element of {0, 1}*, let ¬*w* be the string that is derived from *w* by replacing every 0 by 1 and every 1 by 0. So, for example, ¬011 = 100. Consider an infinite sequence *S* defined as follows:

S_0 = 0.
$S_{n+1} = S_n \neg S_n$.

The first several elements of *S* are 0, 01, 0110, 01101001, 0110100110010110. Describe a Turing machine *M* to output *S*. Assume that *M* starts with its tape equal to $\underline{\theta}$. Also assume the existence of the printing subroutine *P*, defined in Section 20.5.1, but now with one small change: if *M* is a multitape machine, *P* will output the value of tape 1. (Hint: use two tapes.)

1. Write 0 on tape 1.
2. Do forever:
   2.1. *P*.
   2.2. Moving from left to right along tape 1, copy ¬tape 1 to tape 2.
   2.3. Moving from left to right, append tape 2 to tape 1.
   2.4. Erase tape 2.

11) Recall the function *mix*, defined in Example 8.23. Neither the regular languages nor the context-free languages are closed under *mix*. Are the decidable languages closed under *mix*? Prove your answer.

The decidable languages are closed under *mix*. If L is a decidable language, then it is decided by some Turing machine M. We construct a new Turing machine M' that accepts *mix*(L) and that works as follows on input x:

1. Check to see that x has even length. If it does not, reject.
2. Find the middle of x.
3. Reverse the second half of it.
4. Invoke M. If it accepts, accept. If it rejects, reject.

So M' accepts x iff $x = yz^R$, $|y| = |z|$, and $yz \in L$. So accepts *mix*(L). Since there is a Turing machine that accepts *mix*(L), it must be decidable. So the decidable languages are closed under *mix*.

12) Let $\Sigma = \{a, b\}$. Consider the set of all languages over Σ that contain only even length strings.

a) How many such languages are there?

Uncountably infinitely many. The set of even length strings of a's and b's is countably infinite. So its power set is uncountably infinite.

b) How many of them are semidecidable?

Countably infinitely many. The set of all semidecidable languages is countably infinite because the number of Turing machines is countably infinite. So that's an upper bound. And all of the following languages are SD: {aa}, {aaaa}, {aaaaaa}, … That set is countably infinite. So that's a lower bound.

13) Show that every infinite semidecidable language has a subset that is not decidable.

Let L be any infinite language. It has an uncountably infinite number of subsets. There are only countably infinitely many decidable languages (since there are only countably infinitely many Turing machines). So an uncountably infinite number of L's subsets must not be decidable.

21 Decidability and Undecidability Proofs

1) For each of the following languages L, state whether it is in D, in SD/D, or not in SD. Prove your answer. Assume that any input of the form $\langle M \rangle$ is a description of a Turing machine.

a) {a}.

D. L is finite and thus regular and context-free. By Theorem 20.1, every context-free language is in D.

b) $\{\langle M \rangle : \mathtt{a} \in L(M)\}$.

SD/D. Let R be a mapping reduction from H to L defined as follows:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x, operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Return $\langle M\# \rangle$.

If *Oracle* exists, then C = *Oracle*($R(\langle M, w \rangle)$) decides L:

- R can be implemented as a Turing machine.
- C is correct: $M\#$ accepts everything or nothing, depending on whether M halts on w. So:
  - $\langle M, w \rangle \in H$: M halts on w, so $M\#$ accepts all inputs, including a. *Oracle* accepts.
  - $\langle M, w \rangle \notin H$: M does not halt on w, so $M\#$ accepts nothing. In particular, it does not accept a. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

c) $\{\langle M \rangle : L(M) = \{\mathtt{a}\}\}$.

¬SD: Let R be a reduction from ¬H = $\{\langle M, w \rangle$: TM M does not halt on $w\}$ to L, defined as follows:

$R(\langle M, w \rangle) =$

1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. If x = a, accept.
   1.2. Erase the tape.
   1.3. Write w.
   1.4. Run M on w.
   1.5. Accept.
2. Return $\langle M\# \rangle$.

If *Oracle* exists and semidecides L, then $C = R(\langle M, w \rangle)$ semidecides ¬H:

- $\langle M, w \rangle \in$ ¬H: M does not halt on w, so $M\#$ accepts the string a and nothing else. So $L(M\#) = \{\mathtt{a}\}$. *Oracle* accepts.
- $\langle M, w \rangle \notin$ ¬H: M halts on w. $M\#$ accepts everything. So $L(M\#) \neq \{\mathtt{a}\}$. *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

d) $\{<M_a, M_b> : \varepsilon \in L(M_a) - L(M_b)\}$.

¬SD. Let R be a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows: $R(<M, w>)$ =

1. Construct the description of $M\#(x)$ that operates as follows:
   1.1. Erase the tape.
   1.2. Write w.
   1.3. Run M on w.
   1.4. Accept.
2. Construct the description of $M?(x)$ that, on input x, operates as follows:
   2.1. Accept.
3. Return $<M?, M\#>$.

If *Oracle* exists and semidecides L, then C = *Oracle*($R(<M, w>)$) semidecides ¬H:

- R can be implemented as a Turing machine.
- C is correct: $M?$ accepts everything, including ε. $M\#$ accepts everything or nothing, depending on whether M halts on w. So:
  - $<M, w> \in$ ¬H: M does not halt on w. $M\#$ gets stuck in step 1.3. $L(M\#) = \varnothing$. $L(M?) - L(M\#) = L(M?)$, which contains ε. So *Oracle* accepts.
  - $<M, w> \notin$ ¬H: M halts on w. So $L(M\#) = \Sigma^*$. $L(M?) - L(M\#) = \varnothing$, which does not contain ε. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

e) $\{<M_a, M_b> : L(M_a) = L(M_b) - \{\varepsilon\}\}$.

¬SD.

f) $\{<M_a, M_b> : L(M_a) \neq L(M_b)\}$.

¬SD. Let R be a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows: $R(<M, w>)$ =

1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. Erase the tape.
   1.2. Write w.
   1.3. Run M on w.
   1.4. Accept.
2. Construct the description of $M?(x)$ that, on input x, operates as follows:
   2.1. Accept.
3. Return $<M?, M\#>$.

If *Oracle* exists and semidecides L, then C = *Oracle*($R(<M, w>)$) semidecides ¬H:

- R can be implemented as a Turing machine.
- C is correct: $L(M?) = \Sigma^*$. $M\#$ accepts everything or nothing, depending on whether M halts on w. So:
  - $<M, w> \in$ ¬H: M does not halt on w. $M\#$ gets stuck in step 1.3. $L(M\#) = \varnothing$. $L(M?) \neq L(M\#)$. So *Oracle* accepts.
  - $<M, w> \notin$ ¬H: M halts on w. So $L(M\#) = \Sigma^*$. $L(M?) = L(M\#)$. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

g) $\{<M, w> : M$, when operating on input w, never moves to the right on two consecutive moves$\}$.

D. Notice that $M = (K, \Sigma, \Gamma, \delta, s, H)$ must move either to the right or the left on each move. If it cannot move right on two consecutive moves, then every time it moves right, it must next move back left. So it will never be able to read more than the first square of its input tape. It can, however, move left indefinitely. That part of the tape is already known to contain only blanks. M can write on the tape as it

moves left, but it cannot ever come back to read anything that it has written except the character it just wrote and the one immediately to its right. So the rest of the tape is no longer an effective part of *M*'s configuration. We need only consider the current square and one square on either side of it. Thus the number of effectively distinct configurations of *M* is $max = |K| \cdot |\Gamma|^3$. Once *M* has executed *max* steps, it must either halt or be in a loop. If the latter, it will just keep doing the same thing forever. So the following procedure decides *L*:

Run *M* on *w* for $|K| \cdot |\Gamma|^3 + 1$ moves or until *M* halts or moves right on two consecutive moves:
- If *M* ever moves right on two consecutive moves, halt and reject.
- If *M* halts without doing that or if it has not done that after $|K| \cdot |\Gamma|^3 + 1$ moves, halt and accept.

h) {<*M*> : *M* is the only Turing machine that accepts *L*(*M*)}.

D. $L = \varnothing$, since any language that is accepted by some Turing machine is accepted by an infinite number of Turing machines.

i) {<*M*> : *L*(*M*) contains at least two strings}.

SD/D: The following algorithm semidecides *L*:

Run *M* on the strings in Σ* in lexicographic order, interleaving the computations. As soon as two such computations have accepted, halt.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM M halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M*.
   1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:
- <*M*, *w*> ∈ H: *M* halts on *w* so *M*# accepts everything and thus accepts at least two strings, so *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M*# doesn't halt and thus accepts nothing and so does not accept at least two strings so *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

j) {<*M*> : *M* rejects at least two even length strings}.

SD/D: The following algorithm semidecides *L*:

Run *M* on the even length strings in Σ* in lexicographic order, interleaving the computations. As soon as two such computations have rejected, halt.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M*.
   1.4. Reject.
2. Return <*M*#>.

If *Oracle* exists and decides L, then $C = Oracle(R(\langle M, w\rangle))$ decides H:

- $\langle M, w\rangle \in$ H: M halts on w so $M\#$ rejects everything and thus rejects at least two even length strings, so *Oracle* accepts.
- $\langle M, w\rangle \notin$ H: M doesn't halt on w so $M\#$ doesn't halt and thus rejects nothing and so does not reject at least even length two strings. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

k) $\{\langle M\rangle : M \text{ halts on all palindromes}\}$.

¬SD: Assume that $\Sigma \neq \varnothing$. R is a reduction from ¬H = $\{\langle M, w\rangle$: TM M does not halt on $w\}$ to L, defined as follows:

$R(\langle M, w\rangle) =$

1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. Save its input x on a second tape.
   1.2. Erase the tape.
   1.3. Write w.
   1.4. Run M on w for $|x|$ steps or until it halts.
   1.5. If M would have halted, then loop.
   1.6. Else halt.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and semidecides L, then $C = R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: M does not halt on w, so $M\#$ always gets to step 1.6. So it halts on everything, including all palindromes, so *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: M halts on w. Suppose it does so in k steps. Then, for all strings of length k or more, $M\#$ loops at step 1.5. For any k, there is a palindrome of length greater than k. So $M\#$ fails to accept all palindromes. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

l) $\{\langle M\rangle : L(M) \text{ is context-free}\}$.

¬SD. R is a reduction from ¬H = $\{\langle M, w\rangle$: TM M does not halt on $w\}$ to L, defined as follows:

$R(\langle M, w\rangle) =$

1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. Save x.
   1.2. Erase the tape.
   1.3. Write w.
   1.4. Run M on w.
   1.5. If $x \in A^nB^nC^n$, accept. Else loop.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and semidecides L, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: M does not halt on w. $M\#$ gets stuck in step 1.4. So $L(M\#) = \varnothing$, which is context-free. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: M halts on w. So $L(M\#) = A^nB^nC^n$, which is not context-free. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

m) $\{<M> : L(M)$ is not context-free$\}$.

¬SD. R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. If $x \in A^nB^nC^n$, accept.
   1.2. Erase the tape.
   1.3. Write w.
   1.4. Run M on w.
   1.5. Accept.
2. Return $<M\#>$.

If *Oracle* exists and semidecides L, then C = *Oracle*$(R(<M, w>))$ semidecides ¬H:
- $<M, w> \in$ ¬H: M does not halt on w. $M\#$ gets stuck in step 1.4. So $L(M^*) = A^nB^nC^n$, which is not context-free. So *Oracle* accepts.
- $<M, w> \notin$ ¬H: M halts on w. So $L(M\#) = \Sigma^*$, which is context -free. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

n) $\{<M> : A_{\#}(L(M)) > 0\}$, where $A_{\#}(L) = |L \cap \{\mathtt{a}^*\}|$.

SD/D: The following algorithm semidecides L: Lexicographically enumerate the strings in a* and run them through M in dovetailed mode. If M ever accepts a string, accept.

We show not in D by reduction: R is a reduction from H to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1 Erase the tape.
   1.2 Write w.
   1.3 Run M on w.
   1.4 Accept.
2. Return $<M\#>$.

If *Oracle* exists and decides L, then C = *Oracle*$(R(<M, w>))$ decides H:
- $<M, w> \in$ H: M halts on w. $M\#$ accepts everything, including strings in a*. So *Oracle* accepts.
- $<M, w> \notin$ ¬H: M does not halt on w. $M\#$ accepts nothing. So *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

o) $\{<M> : |L(M)|$ is a prime integer greater than $0\}$.

¬SD: Assume that $\Sigma \neq \varnothing$. R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. If x is one of the first two strings (lexicographically) in Σ^*, accept.
   1.2. Erase the tape.
   1.3. Write w on the tape.
   1.4. Run M on w.
   1.5. Accept.
2. Return $<M\#>$.

If *Oracle* exists and semidecides L, then $C = R(\langle M, w\rangle)$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: M does not halt on w, so the $M\#$ accepts only the two strings that it accepts in step 1.1. So $|L(M\#)| = 2$, which is greater than 0 and prime, so *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: M halts on w so, $M\#$ accepts everything else at step 1.5. There is an infinite number of strings over any nonempty alphabet, so $L(M\#)$ is infinite. Its cardinality is not a prime integer. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

p) $\{\langle M\rangle$: there exists a string w such that $|w| < |\langle M\rangle|$ and that M accepts $w\}$.

SD/D: The following algorithm semidecides L:

Run M on the strings in Σ^* of length less than $|\langle M\rangle|$, in lexicographic order, interleaving the computations. If any such computation halts, halt and accept.

Proof not in D: R is a reduction from H = $\{\langle M, w\rangle$: TM M halts on $w\}$ to L, defined as follows:

$R(\langle M, w\rangle)$ =
1. Construct the description $\langle M\#\rangle$ of a new Turing machine $M\#(x)$ that, on input x, operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and decides L, then C = *Oracle*$(R(\langle M, w\rangle))$ decides H:

- $\langle M, w\rangle \in$ H: M halts on w so $M\#$ accepts everything. So, in particular, it accepts ε, which is a string of length less than $|\langle M\rangle|$, so *Oracle*$(\langle M\#\rangle)$ accepts.
- $\langle M, w\rangle \notin$ H: M doesn't halt on w so $M\#$ doesn't halt and thus accepts nothing. So, in particular there is no string of length less than $|\langle M\rangle|$ that $M\#$ accepts, so *Oracle*$(\langle M\#\rangle)$ rejects.

But no machine to decide H can exist, so neither does *Oracle*.

q) $\{\langle M\rangle$: M does not accept any string that ends with `0`$\}$.

¬SD: R is a reduction from ¬H = $\{\langle M, w\rangle$: TM M does not halt on $w\}$ to L, defined as follows:

$R(\langle M, w\rangle)$ =
1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and semidecides L, then $C = R(\langle M, w\rangle)$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: M does not halt on w so $M\#$ accepts nothing and so, in particular, accepts no string that ends in `0`. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: M halts on w so $M\#$ accepts everything, including all strings that end in `0`. Since $M\#$ does accept strings that end in `0`, $M\#$ is not in L and *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

r) {<*M*> : there are at least two strings *w* and *x* such that *M* halts on both *w* and *x* within some number of steps *s*, and *s* < 1000 and *s* is prime}.

D. Note that in any fixed number *s* steps, *M* can examine no more than *s* squares of its tape. So if it is going to accept any string *w* that is longer than *s*, it must also accept a string *w′* that is no longer than *s* and that is an initial substring of *w*. So the following algorithm decides *L*:

Run *M* on all strings in Σ* of length between 0 and 1000. Try each for 1000 steps or until the computation halts:

- If at least two such computations halted in some prime number of steps *s*, accept.
- Else reject.

s) {<*M*> : there exists an input on which TM *M* halts in fewer than |<*M*>| steps}.

D. In |<*M*>| steps, *M* can examine no more than |<*M*>| squares of its tape. So the following algorithm decides *L*:

Run *M* on all strings in Σ* of length between 0 and |<*M*>|. Try each for |<*M*>| steps or until the computation halts:

- If at least one such computation halted, accept.
- Else reject.

It isn't necessary to try any longer strings because, if *M* accepts some longer string, it does so by looking at no more than |<*M*>| initial characters. So it would also accept the string that contains just those initial characters. And we'd have discovered that.

t) {<*M*> : *L*(*M*) is infinite}.

¬SD. Assume that Σ ≠ ∅. R is a reduction from ¬H = {<*M*, *w*> : TM M does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Save its input *x* on a second tape.
   1.2. Erase the tape.
   1.3. Write *w*.
   1.4. Run *M* on *w* for |*x*| steps or until it halts.
   1.5. If *M* would have halted, then loop.
   1.6. Else accept.
2. Return *Oracle*(<*M*#>)

If *Oracle* exists and semidecides *L*, then *R* semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*. So *M*# always makes it to step 1.6. It accepts everything, which is an infinte set. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then *M*# loops on all strings of length *k* or greater. It accepts strings of length less than *k*. But that set is finite. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

u) {<*M*> : *L*(*M*) is uncountably infinite}.

D. *L* = ∅, since every Turing machine *M* = (*K*, Σ, Γ, δ, *s*, *H*) accepts some subset of Σ* and |Σ*| is countably infinite. So *L* is not only in D, it is regular.

v) {<*M*> : TM *M* accepts the string <*M*, *M*> and does not accept the string <*M*>}.

¬SD. *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
    1.1. If *x* is of the form *yy* for some *y*, accept.
    1.2. Erase the tape.
    1.3. Write *w* on the tape.
    1.4. Run *M* on *w*.
    1.5. Accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *R*(<*M*, *w*>)) semidecides ¬H:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*. So *M*# accepts <*M, M*> and does not accept anything that is not of the form *yy*, including <*M*> (which can never be of that form). So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. So *M*# accepts everything. It thus accepts <*M*>. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

w) {<*M*> : TM *M* accepts at least two strings of different lengths}.

SD/D: The following algorithm semidecides *L*:

Lexicographically enumerate the strings over Σ_M^* and run *M* on them in interleaved mode. Each time *M* accepts a string *s*, record |*s*|. If *M* ever accepts two strings of different lengths, halt and accept.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M* on *w*.
    1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:
- <*M*, *w*> ∈ H: *M* halts on *w* so *M*# accepts everything. So it accepts at least two strings of different lengths. *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M*# accepts nothing. *M*# does not accept two strings of any length, so *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

x) {<*M*> : TM *M* accepts exactly two strings and they are of different lengths}.

¬SD: Assume that Σ ≠ ∅. *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
    1.1. If *x* = `a` or *x* = `aa` accept.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w*.
    1.5. Accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M*# accepts at least two strings (`a` and `aa`) of different lengths. *M* does not halt on *w*, so, on all other inputs, *M*# gets stuck in step 1.4. So *M*# accepts exactly two strings and they are of different lengths, so *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M*# accepts everything. So it accepts more than two strings. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

y) {<*M*, *w*> : TM *M* accepts *w* and rejects w^R}.

SD/D: The following algorithm semidecides *L*:

Run *M* on *w*. If it accepts, run *M* on w^R. If it rejects, accept. In all other cases, loop.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Save *x*.
   1.2. Erase the tape.
   1.3. Write *w* on the tape.
   1.4. Run *M* on *w*.
   1.5. If *x* = `ba` then reject.
   1.6. Else accept.
2. Return <*M*#, `ab`>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- <*M*, *w*> ∈ H: *M* halts on *w* so *M*# always makes it to step 1.5. It accepts `ab` and rejects `ba` so *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M*# accepts nothing, so *Oracle* does not accept.

But no machine to decide H can exist, so neither does *Oracle*.

z) {<*M*, *x*, *y*> : *M* accepts *xy*}.

SD/D: The following algorithm semidecides *L*:

Run *M* on *xy*. If it accepts, accept.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M*#, ε, ε>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*>)) decides H. *R* can be implemented as a Turing machine. And *C* is correct. *M*# accepts everything or nothing, depending on whether *M* halts on *w*. So:

- <*M*, *w*> ∈ H: *M* halts on *w* so *M*# accepts everything, including εε, so *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M*# accepts nothing, including εε, so *Oracle* rejects.

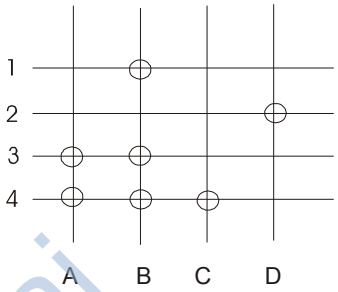
But no machine to decide *H* can exist, so neither does *Oracle*.

aa) {<*D*> : <*D*> is the string encoding of a deterministic FSM *D* and $L(D) = \varnothing$}.

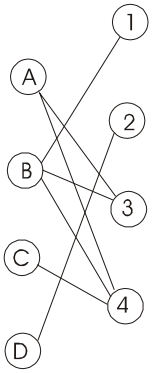
D. The following algorithm decides *L*:

1. Check to see that <*D*> is the string encoding of some deterministic FSM. If it is not, reject.
2. Else use *emptyFSM* to decide whether $L(D) = \varnothing$. If it is, accept. Else reject.

2) In E.3, we describe a straightforward use of reduction that solves a grid coloring problem by reducing it to a graph problem. Given the grid *G* shown here:



a) Show the graph that corresponds to *G*.



b) Use the graph algorithm we describe to find a coloring of *G*.

A, 3, B, 4, A is a cycle. Color A-3 red, B-3 blue, B-4 red, and A-4 blue and then remove those edges from the graph.. That leaves the edges B-1, C-4, and D-2. These edges correspond to independent subtrees that can be colored independently. So color them all red.

3) In this problem, we consider the relationship between H and a very simple language {a}.

a) Show that {a} is *mapping* reducible to H.

We must show a computable function *R* that maps instances of {a} to instances of H. Define *R* as follows:

R(*w*) =

1. If $w \in$ {a}, construct the description of *M*#(*x*) that immediately halts on all inputs.
2. If $w \notin$ {a}, construct the description of *M*#(*x*) that immediately loops on all inputs.
3. Return <*M*#, ε>.

If *Oracle* exists and decides H, then *C* = *Oracle*(*R*(*w*)) decides {a}:

- $s \in$ {a}: *M*# halts on everything, including ε, so *Oracle*(<*M*#, ε>) accepts.
- $s \notin$ {a}: *M*# halts on nothing, including ε, so *Oracle*(<*M*#, ε>) rejects.

b) Is it possible to reduce H to {a}? Prove your answer.

No. The language {a} is in D. (In fact, it is regular.) So, if there were a reduction from H to {a} then H would be in D. But it is not. So no such reduction can exist.

4) Show that H_{ALL} is not in D by reduction from H.

Let *R* be a mapping reduction from H to H_{ALL} defined as follows:

R(< *M*, *w*>) =
1. Construct the description <*M*#> of a new Turing machine *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
2. Return <*M*#>.

If *Oracle* exists and decides H_{ALL}, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H. *R* can be implemented as a Turing machine. And *C* is correct. *M*# ignores its own input. It halts on everything or nothing. So:

- < *M*, *w*> ∈ *H*: *M* halts on *w*, so *M*# halts on all inputs. *Oracle* accepts.
- < *M*, *w*> ∉ *H*: *M* does not halt on *w*, so *M*# halts on nothing. *Oracle* rejects.

But no machine to decide *H* can exist, so neither does *Oracle*.

5) Show that each of the following languages is not in D:

a) $A_\varepsilon = \{<M> : \text{TM } M \text{ accepts } \varepsilon\}$.

We show that A_ε is not in D by reduction from H. Let *R* be a mapping reduction from H to A_ε defined as follows:

R(<*M*, *w*>) =
1. Construct the description <*M*#> of a new Turing machine *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides A_ε, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H. *R* can be implemented as a Turing machine. And *C* is correct. *M*# ignores its own input. It halts on everything or nothing. So:

- <*M*, *w*> ∈ H: *M* halts on *w*, so *M*# accepts everything, including ε. *Oracle*(<*M*#>) accepts.
- <*M*, *w*> ∉ H: *M* does not halt on *w*, so *M*# halts on nothing. So it does not accept ε. *Oracle*(<*M*#>) rejects.

But no machine to decide H can exist, so neither does *Oracle*.

b) $A_{ANY} = \{<M> : \text{TM } M \text{ accepts at least one string}\}$.

We show that A_{ANY} is not in D by reduction from H. Let *R* be a mapping reduction from H to A_{ANY} defined as follows:

R(<*M*, *w*>) =
1. Construct the description <*M*#> of a new Turing machine *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides A_{ANY}, then C = *Oracle*(R(<M, w>)) decides H. R can be implemented as a Turing machine. And C is correct. M# ignores its own input. It halts on everything or nothing. So:

- <M, w> ∈ H: M halts on w, so M# accepts everything. So it accepts at least one string. *Oracle*(<M#>) accepts.
- <M, w> ∉ H: M does not halt on w, so M# halts on nothing. So it does not accept even one string. *Oracle*(<M#>) rejects.

But no machine to decide H can exist, so neither does *Oracle*.

c) A_{ALL} = {<M> : = $L(M) = \Sigma_M^*$ }.

d) {<M, w> : Turing machine M rejects w}.

e) {<M, w> : Turing machine M is a deciding TM and M rejects w}.

6) Show that L = {<M> : Turing machine M, on input ε, ever writes `0` on its tape} is in D iff H is in D. In other words, show that $L \leq$ H and H $\leq L$.

We first show that $L \leq$ H. Define the following reduction:

R(<M>) =
1. Construct the description <M#> of a new Turing machine M#(x) that, on input x, operates as follows:
   1.1. Run M on ε until it writes a `0` or halts.
   1.2. If it halted before writing a `0`, then loop.
2. Return <M#, ε>.

If *Oracle* exists and decides H, then C = *Oracle*(R(<M>)) decides L. R can be implemented as a Turing machine. And C is correct. M# ignores its own input. So:

- <M> ∈ L: M writes a `0` on input ε. M# haltss everything, including ε. *Oracle*(<M#, ε>) accepts.
- <M> ∉ L: M does not write a `0` on input ε. Then there are two cases. If M runs forever without writing a `0`, M# will loop forever in step 1.1. If M halts without writing a `0`, M# will loop forever in step 1.2. So M# halts on nothing. So it does not accept ε. *Oracle*(<M#, ε>) rejects.

Next we show that H $\leq L$. Define the following reductions:

R(<M, w>) =
1. Construct the description <M#> of a new Turing machine M#(x) that, on input x, operates as follows:
   1.1. If `0` is in M's tape alphabet, change M and w to substitute for it some other symbol that was not already the alphabet.
   1.2. Erase the tape.
   1.3. Write w on the tape.
   1.4. Run M on w.
   1.5. Write `0`.
2. Return <M#>.

If *Oracle* exists and decides A_{ANY}, then C = *Oracle*(R(<M, w>)) decides H. R can be implemented as a Turing machine. And C is correct. M# ignores its own input. It halts on everything or nothing. So:

- <M, w> ∈ H: M halts on w. So, on all inputs, including ε, M# makes it to step 1.5 and writes a `0`. *Oracle*(<M#>) accepts.
- <M, w> ∉ H: M does not halt on w. So, on all inputs, including ε, M# gets stuck in step 1.4. Since M has been modified to guarantee that it never writes a `0`, M# never writes a `0`. *Oracle*(<M#>) rejects.

7) Show that each of the following questions is undecidable by recasting it as a language recognition problem and showing that the corresponding language is not in D:

a) Given a program P, input x, and a variable n, does P, when running on x, ever assign a value to n?

$L = \{<P, x, n> : P$, when running on x, ever assigns a value to $n\}$. We show that L is not in D by reduction from H. Define:

$R(<M, w>) =$

1. Construct the description $<P>$ of a program P that ignores its input and operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Set n to 0.
2. Return $<P, \varepsilon, n>$.

If *Oracle* exists and decides L, then $C = Oracle(R(<M, w>))$ decides H. R can be implemented as a Turing machine. And C is correct:

- $<M, w> \in$ H: M halts on w, so P, regardless of its input, assigns a value to n. $Oracle(<P, \varepsilon, n>)$ accepts.
- $<M, w> \notin$ H: M does not halt on w, so P, regardless of its input, fails to assign a value to n. . $Oracle(<P, \varepsilon, n>)$ rejects.

But no machine to decide H can exist, so neither does *Oracle*.

b) Given a program P and code segment S in P, does P reach S on every input (in other words, can we guarantee that S happens)?

c) Given a program P and a variable x, is x always initialized before it is used?

d) Given a program P and a file f, does P always close f before it exits?

e) Given a program P with an array reference of the form $a[i]$, will i, at the time of the reference, always be within the bounds declared for the array?

f) Given a program P and a database of objects d, does P perform the function f on all elements of d?

$L = \{<P, d, f> : P$ performs f on every element of $d\}$. We show that L is not in D by reduction from H. Define:

$R(<M, w>) =$

1. Create a database D with one record r.
2. Create the function f that writes the value of the first field of the database object it is given.
3. Construct the description $<P>$ of a program P that ignores its input and operates as follows:
   3.1. Erase the tape.
   3.2. Write w on the tape.
   3.3. Run M on w.
   3.4. Run f on r.
4. Return $<P, D, f>$.

If *Oracle* exists and decides *L*, then $C = Oracle(R(\langle M, w\rangle))$ decides H. *R* can be implemented as a Turing machine. And *C* is correct:

- $\langle M, w\rangle \in$ H: *M* halts on *w*, so *P*, regardless of its input, runs *f* on *r*. $Oracle(\langle P, D, f\rangle)$ accepts.
- $\langle M, w\rangle \notin$ H: *M* does not halt on *w*, so *P*, regardless of its input, fails to run f on *r*. . $Oracle(\langle P, D, f\rangle)$ rejects.

But no machine to decide H can exist, so neither does *Oracle*.

8) Theorem J.1 tells us that the safety of even a very simple security model is undecidable, by reduction from H_ε. Show an alternative proof that reduces $A = \{\langle M, w\rangle : M \text{ is a Turing machine and } w \in L(M)\}$ to the language Safety.

Define:

$R(\langle M, w\rangle) =$

1. Make any necessary changes to *M*. Do this as described in step 1.2 of the reduction given in the proof of Theorem J.1. (Note that, by definition, *M* has only a single state named *y*, so step 1.1 isn't necessary.)
2. Build *S*:
   2.1. Construct an initial access control matrix *A* that corresponds to *M*'s initial configuration on input *w*.
   2.2. Construct a set of commands, as described in the proof of Theorem J.1, that correspond to the transitions of *M*.
3. Return $\langle S, y\rangle$.

$\{R, \neg\}$ is a reduction from A to Safety. If *Oracle* exists and decides Safety, then $C = \neg Oracle(R(\langle M, w\rangle))$ decides A. *R* and ¬ can be implemented as a Turing machines. And *C* is correct. By definition, *S* is unsafe with respect to *y* iff *y* is not present in the initial configuration of *A* and there exists some sequence of commands in *S* that could result in the initial configuration of *S* being transformed into a new configuration in which *y* has leaked, i.e., it appears in some cell of *A*. Since the initial configuration of *S* corresponds to *M* being in its initial configuration on *w*, *M* does not start in *y*, and the commands of *S* simulate the moves of *M*, this will happen iff *M* reaches state *y* and so accepts. Thus:

- If $\langle M, w\rangle \in$ A: *M* accepts *w*, so *y* eventually appears in some cell of *A*. *S* is unsafe with respect to *y*, so *Oracle* rejects. *C* accepts.
- If $\langle M, w\rangle \notin$ A: *M* does not accept *w*, so *y* never appears in some cell of *A*. *S* is safe with respect to *y*, so *Oracle* accepts. *C* rejects.

But no machine to decide A can exist, so neither does *Oracle*.

9) Show that each of the following languages *L* is not in SD:

a) $\neg H_\varepsilon$.

b) EqTMs.

$A_{ALL} \leq_M$ EqTMs and so EqTMs is not in SD:

Let $R(\langle M\rangle) =$

1. Construct the description $\langle M\#\rangle$ of a new machine $M\#(x)$ that, on input *x*, operates as follows:
   1.1. Accept.
2. Return $\langle M, M\#\rangle$.

If *Oracle* exists and semidecides CompTMs, then $C = Oracle(R(\langle M \rangle))$ semidecides A_{ALL}:

- *R* can be implemented as a Turing machine
- *C* is correct: *M*# ignores its own input. It accepts everything. So:
  - $\langle M \rangle \in A_{ALL}$: *M*# accepts everything and so does *M*. *Oracle* accepts.
  - $\langle M \rangle \notin A_{ALL}$: *M*# accepts everything but *M* does not. *Oracle* does not accept.

But no machine to semidecide A_{ALL} can exist, so neither does *Oracle*.

c) TM_{REG}.

$\neg H \leq_M TM_{REG}$ and so TM_{REG} is not in SD:

Let $R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input *x*, operates as follows:
   1.1. Copy its input *x* to a second tape.
   1.2. Erase the tape.
   1.3. Write *w* on the tape.
   1.4. Run *M* on *w*.
   1.5. Put *x* back on the first tape.
   1.6. If $x \in A^nB^n$ then accept, else reject.
2. Return $\langle M\# \rangle$.

R is a reduction from ¬H to TM_{REG}. If *Oracle* exists and semidecides TM_{REG}, then $C = Oracle(R(\langle M, w \rangle))$ decides H:

- *R* can be implemented as a Turing machine
- *C* is correct:
  - $\langle M, w \rangle \in \neg H$: *M* does not halt on *w*. *M*# gets stuck in step 1.4 and so accepts nothing. $L(M\#) = \varnothing$, which is regular. $Oracle(\langle M\# \rangle)$ accepts.
  - $\langle M, w \rangle \notin \neg H$: *M* halts on *w*, so *M*# makes it to step 1.5. Then it accepts *x* iff $x \in A^nB^n$. So *M*# accepts A^nB^n, which is not regular. $Oracle(\langle M\# \rangle)$ does not accept.

But no machine to decide H can exist, so neither does *Oracle*.

d) $\{\langle M \rangle : |L(M)| \text{ is even}\}$.

$\neg H \leq_M L$ and so *L* is not in SD:

Let $R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new machine $M\#(x)$ that, on input *x*, operates as follows:
   1.1. Copy the input *x* to a second tape.
   1.2. Erase the tape.
   1.3. Write *w* on the tape.
   1.4. Run *M* on *w*.
   1.5. Put *x* back on the tape.
   1.6. If $x = \varepsilon$ then accept; else loop.
2. Return $\langle M\# \rangle$.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w \rangle))$ semidecides ¬H:

- *R* can be implemented as a Turing machine.
- *C* is correct:
  - $\langle M, w \rangle \in \neg H$: $L(M\#) = \varnothing$. $|\varnothing| = 0$,which is even. *Oracle* accepts.
  - $\langle M, w \rangle \notin \neg H$: $L(M\#) = \{\varepsilon\}$. $|\{\varepsilon\}| = 1$, which is odd. *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

e) $\{\langle M \rangle$: Turing machine *M* accepts all even length strings$\}$.

$\neg H \leq_M L$ and so L is not in SD:

Let $R(\langle M \rangle) =$
1. Construct the description $\langle M\# \rangle$ of a new machine $M\#(x)$ that, on input x, operates as follows:
   1.1. Copy the input x to a second tape.
   1.2. Erase the tape.
   1.3. Write w on the tape.
   1.4. Run M on w for $|x|$ steps or until M naturally halts.
   1.5. If M halted naturally, then loop. Else accept.
2. Return $\langle M\# \rangle$.

If *Oracle* exists and semidecides L, then $C = Oracle(R(\langle M, w \rangle))$ semidecides $\neg H$:
- R can be implemented as a Turing machine.
- C is correct:
  - $\langle M \rangle \in \neg H$: M will never halt naturally. So $M\#$ accepts everything, including all even length strings. *Oracle* accepts.
  - $\langle M \rangle \notin \neg H$: M may not halt naturally on some number of short strings. So $M\#$ will accept them. But there is an infinite number of "longer" strings (i.e., those whose length is at least as long as the number of steps M executes before it halts), on which the simulation of M will halt naturally. $M\#$ will loop on them. Some of them will be of even length. So $M\#$ does not accept all even length strings. *Oracle* does not accept.

But no machine to semidecide $\neg H$ can exist, so neither does *Oracle*.

f) $\{\langle M \rangle$: Turing machine M accepts no even length strings$\}$.

$\neg H \leq_M L$ and so L is not in SD:

Let $R(\langle M, w \rangle) =$
1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x, operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Return $\langle M\# \rangle$.

If *Oracle* exists and semidecides L, then $C = Oracle(R(\langle M, w \rangle))$ decides $\neg H$. $M\#$ ignores its own input. It accepts everything or nothing. So:

- $\langle M, w \rangle \in \neg H$: M does not halt on w, so $M\#$ accepts nothing. So it accepts no even length strings. $Oracle(\langle M\# \rangle)$ accepts.
- $\langle M, w \rangle \notin \neg H$: M halts on w, so $M\#$ accepts everything. So it accepts all even length strings. $Oracle(\langle M\# \rangle)$ does not accept.

But no machine to decide H can exist, so neither does *Oracle*.

g) $\{\langle M \rangle$: Turing machine M does not halt on input $\langle M \rangle\}$.

$\neg H \leq_M L$ and so L is not in SD:

Let $R(\langle M, w \rangle) =$
1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x, operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
2. Return $\langle M\# \rangle$.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) decides ¬H. M# ignores its own input. It halts on everything or nothing. So:

- <M, w> ∈ ¬H: M does not halt on w, so M# halts on nothing. So, in particular, it does not halt on input <M>. *Oracle*(<M#>) accepts.
- <M, w> ∉ ¬H: M halts on w, so M# accepts everything. So it does halt on input <M>. *Oracle*(<M#>) does not accept.

But no machine to decide H can exist, so neither does *Oracle*.

h) {<M, w> : M is a deciding Turing machine and M rejects w}.

Note that if all we cared about was whether M rejects w, L would be in SD. But we can't semidecide the question of whether M is a deciding TM.

$A_{ALL} \leq_M L$ and so L is not in SD:

Let R(<M>) =
1. Construct the description <M#> of a new machine M#(x) that, on input x, operates as follows:
   1.1. Run M on x.
   1.2. Reject.
2. Return <M#, ε>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M>)) semidecides A_{ALL}:

- R can be implemented as a Turing machine
- C is correct: M# ignores its own input. It accepts everything. So:
  - <M> ∈ A_{ALL}: M# halts on all inputs. It also rejects all inputs, including ε. *Oracle* accepts.
  - <M> ∉ A_{ALL}: There is at least one input on which M (and thus M#) doesn't halt. So M# is not a deciding TM. *Oracle* does not accept.

But no machine to semidecide A_{ALL} can exist, so neither does *Oracle*.

10) Do the other half of the proof of Rice's Theorem, i.e., show that the theorem holds if $P(\varnothing)$ = *True*.

The easiest way to do this is to use a reduction that is not a mapping reduction. We simply invert the reduction that we did to prove the first half. So we proceed as follows. Assume that $P(\varnothing)$ = *True*. Since P is nontrivial, there is some SD language L_F such that $P(L_F)$ is *False*. Since L_F is in SD, there exists some Turing machine K that semidecides it.

Define:

R(<M, w>) =
1. Construct the description <M#> of a new Turing machine M#(x) that, on input x, operates as follows:
   1.1. Copy its input x to a second tape.
   1.2. Erase the tape.
   1.3. Write w on the tape.
   1.4. Run M on w.
   1.5. Put x back on the first tape and run K on x.
2. Return <M#>.

{R, ¬} is a reduction from H to L_2. If *Oracle* exists and decides L, then C = ¬*Oracle*(R(<M, w>)) decides H. R can be implemented as a Turing machine. And C is correct:

- If <M, w> ∈ H: M halts on w, so M# makes it to step 1.5. So M# does whatever K would do. So L(M#) = L(K) and P(L(M#)) = P(L(K)). We chose K precisely to assure that P(L(K)) is *False*, so P(L(M#)) must also be *False*. *Oracle* decides P. *Oracle*(<M#>) rejects so C accepts.

- If <*M*, *w*> ∉ H: *M* does not halt on *w*. *M*# gets stuck in step 1.4 and so accepts nothing. *L*(*M*#) = ∅. By assumption, *P*(∅) = *True*. *Oracle* decides *P*. *Oracle*(<*M*#>) accepts so *C* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

11) For each of the following languages *L*, do two things:
   i) State whether or not Rice's Theorem has anything to tell us about the decidability of *L*.
   ii) State whether *L* is in D, SD/D, or not in SD.

a) {<*M*> : *M* accepts all strings that start with `a`}.

Rice's Theorem applies and tells us that *L* is not in D. It is also true that *L* is not in SD.

b) {<*M*> : *M* halts on ε in no more than 1000 steps}.

Rice's Theorem does not apply. *L* is in D. It can be decided by simply running *M* on ε for 1000 steps or until it halts.

c) ¬L_1, where L_1 = {<*M*> : *M* halts on all strings in no more than 1000 steps}.

Rice's Theorem does not apply. L_1 is in D. The key to defining a decision procedure for it is the observation that if *M* is allowed to run for only 1000 steps, it must make its decision about whether to accept an input string *w* after looking at no more than the first 1000 characters of *w*. So we can decide L_1 by doing the following: Lexicographically enumerate the strings of length up 1000 drawn from the alphabet of *M*. For each, run *M* for 1000 steps or until it halts. If *M* halted on all of them, then it must also halt on all longer strings. So accept. Otherwise, reject. Since the decidable languages are closed under complement, ¬L_1 must be in D if L_1.

d) {<*M*, *w*> : *M* rejects *w*}.

Rice's Theorem does not apply. Note that the definition of this language does not ask about the language that *M* accepts. Failure to reject could mean either that M accepts or that it loops. *L* is in SD.

12) Use Rice's Theorem to prove that each of the following languages is not in D:

a) {<*M*> : Turing machine *M* accepts at least two odd length strings}.

We define *P* as follows:
- Let *P* be defined on the set of languages accepted by some Turing machine *M*. Let it be *True* if *L*(*M*) contains at least two odd length strings and *False* otherwise.
- The domain of *P* is the SD languages since it is those languages that are accepted by some Turing machine *M*.
- *P* is nontrivial since *P*({`a`, `aaa`}) is *True* and *P*(∅) is *False*.

Thus {<*M*> : Turing machine *M* accepts at least two odd length strings} is not in D.

b) {<*M*> : *M* is a Turing machine and |*L*(*M*)| = 12}.

We define *P* as follows:
- Let *P* be defined on the set of languages accepted by some Turing machine *M*. Let it be *True* if |*L*(*M*)| is 12 and *False* otherwise.
- The domain of *P* is the SD languages since it is those languages that are accepted by some Turing machine *M*.
- *P* is nontrivial since *P*({`a`, `aa`, `aaa`, `aaaa`, `aaaaa`, `aaaaaa`, `b`, `bb`, `bbb`, `bbbb`, `bbbbb`, `bbbbbb`}) is *True* and *P*(∅) is *False*.

Thus {<*M*> : *M* is a Turing machine and |*L*(*M*)| = 12} is not in D.

13) Prove that there exists no mapping reduction from H to the language $L_2 = \{<M> : \text{Turing machine } M \text{ accepts no even length strings}\}$ that we defined in Theorem 21.9.

Suppose, to the contrary, that there did. Let f be that reduction. Then, by the definition of a mapping reduction, we have that $\forall x\ (x \in \text{H iff } f(x) \in L_2)$. Thus we also have that $\forall x\ (x \in \neg\text{H iff } f(x) \in \neg L_2)$. $\neg L_2$ is $\{<M> : \text{Turing machine } M \text{ accepts at least one even length string}\}$. So we have a reduction from $\neg$H to $\{<M> : \text{Turing machine } M \text{ accepts at least one even length string}\}$. But no such reduction can exist because $\{<M> : \text{Turing machine } M \text{ accepts at least one even length string}\}$ is semidecidable and $\neg$H isn't.

14) Let $\Sigma = \{1\}$. Show that there exists at least one undecidable language with alphabet Σ.

There are countably infinitely many strings drawn from Σ. So there uncountably infinitely many languages containing such strings. There are only countably infinitely many decidable languages (because each must be decided by some Turing machine and there are only a countably infinite number of Turing machines). So there are uncountably infinitely many undecidable languages with alphabet Σ.

15) Give an example of a language L such that neither L nor $\neg L$ is decidable.

Let $L = \{<M> : M \text{ accepts exactly one string}\}$. $\neg L = \{<M> : M \text{ accepts some number of strings other than one}\}$.

16) Let *repl* be a function that maps from one language to another. It is defined as follows:

$repl(L) = \{w : \exists x \in L \text{ and } w = xx\}$.

a) Are the context free languages closed under *repl*? Prove your answer.

No. Let $L = \{a, b\}^*$. Then $repl(L) = WW$, which is not context free.

b) Are the decidable languages closed under *repl*? Prove your answer.

Yes. If L is in D, then there is a Turing machine M that decides it. The following algorithm decides $repl(L)$: On input w do: If $|w|$ is odd, reject. Otherwise, find the middle of w. Run the first half through M. If it rejects, reject. Otherwise, compare the first half of w to the second half. If they are the same, accept. Else reject.

17) For any nonempty alphabet Σ, let L be any decidable language other than $\varnothing$ or Σ^*. Prove that $L \leq_M \neg L$.

Since L is decidable, there is some Turing machine M that decides it. Given any L, and thus M, define the following mapping reduction from L to $\neg L$:

$R(x) =$

1. Run M on x. /*It must halt and either accept or reject.
2. If M accepted, then lexicographically enumerate the strings in Σ^* and run each through M until one is found that M rejects. Return that string. /* Note that there must be such a string since L is not Σ^*.
3. If M rejected, then lexicographically enumerate the strings in Σ^* and run each through M until one is found that M accepts. Return that string. /* Note that there must be such a string since L is not $\varnothing$.

R is a mapping reduction from L to $\neg L$ because $x \in L$ iff $R(x) \in \neg L$.

18) We will say that L_1 is ***doubly reducible*** to L_2, which we will write as $L_1 \leq_D L_2$, iff there exist two computable functions f_1 and f_2 such that:

$\forall x \in \Sigma^*$ (($x \in L_1$) iff ($f_1(x) \in L_2$ and $f_2(x) \notin L_2$)).

Prove or disprove each of the following claims:

a) If $L_1 \leq L_2$ and $L_2 \neq \Sigma^*$, then $L_1 \leq_D L_2$.

We show by counterexample that this claim is false. Let $L_2 = \{<M> : M$ accepts at least one string$\}$. $L_2 \neq \Sigma^*$. H is mapping reducible to L_2 (we omit the details). But H cannot be doubly reducible to L_2 because, if it were, we'd have (by definition) that H $\leq_M \neg L_2$. But then we'd also have that $\neg$H $\leq_M \neg\neg L_2$, and so $\neg$H $\leq_M L_2$. But L_2 is in SD and H isn't, so contradiction.

b) If $L_1 \leq_D L_2$ and $L_2 \in$ D, then $L_1 \in$ D.

This claim is true. If $L_2 \in$ D, then it is decided by some Turing machine. Call it M. Let R be the Turing machine that implements the computable function f_1 that mapping reduces L_1 to L_2. Then $M(R(x))$ decides whether $x \in L_1$. Thus L_1 is in D.

c) For every language L_2, there exists a language L_1 such that $\neg(L_1 \leq_D L_2)$.

This claim is true, which we'll show by contradiction and a counting argument. Assume that the claim were false. In other words, assume that there exists a language L such that, for all languages $L_i \subseteq \Sigma^*$, we have that $L_i \leq_D L$. This means that, for every language $L_i \subseteq \Sigma^*$, there are two computable functions f_1^i and f_2^i such that:

$$\forall x \in \Sigma^* ((x \in L_1) \text{ iff } (f_1^i(x) \in L_2) \text{ and } (f_2^i(x) \notin L_2)). \quad (1)$$

Since there are uncountably many languages, but only countably many computable functions (and thus countably many pairs of computable functions), it follows from (1) and the pigeonhole principle that at least two distinct languages, L_j and L_k, are doubly reducible to L by exactly the same pair of computable functions. But this is true iff $L_j = L_k$ or $L_j = \neg L_k$. If we consider the set X of all distinct languages that are subsets of Σ^*but such that no two languages in X are complementary, X is still uncountably infinite and there are not enough distinct mapping functions. Thus we have a contradiction. Thus, for every language L_2, there exists a language L_1 such that $\neg(L_1 \leq_D L_2)$.

19) Let L_1 and L_2 be any two SD/D languages such that $L_1 \subset L_2$. Is it possible that L_1 is reducible to L_2? Prove your answer.

Yes. Let $L_1 = \{<M, \varepsilon> : M$ accepts $\varepsilon\}$. Let $L_2 =$ H $= \{<M, w> : M$ accepts $w\}$. $L_1 \subset L_2$. We show that $L_1 \leq L_2$ as follows: R is the trivial mapping reduction that, on input $<M, \varepsilon>$ simply returns $<M, \varepsilon>$.

20) If L_1 and L_2 are decidable languages and $L_1 \subseteq L \subseteq L_2$, must L be decidable? Prove your answer.

No. Let $L_1 = \varnothing$. Let $L_2 = \{<M>\}$. Let $L = \{<M> : M$ accepts $\varepsilon\}$, which is not decidable.

21) Goldbach's conjecture states that every even integer greater than 2 can be written as the sum of two primes. (Consider 1 to be prime.) Suppose that A = $\{<M, w> : M$ is a Turing machine and $w \in L(M)\}$ were decidable by some Turing machine *Oracle*. Define the following function:

$G()$ = *True* if Goldbach's conjecture is true,
False otherwise.

Use *Oracle* to describe a Turing machine that computes *G*. You may assume the existence of a Turing machine *P* that decides whether a number is prime.

Define a TM *TryGold* as follows: Lexicographically enumerate the binary encodings of the even integers greater than 2. As each integer is enumerated, see whether it can be written as the sum of two primes by trying all pairs of primes (as determined by *P*) less than it. If it can, continue the loop. If it cannot, halt and accept.

TryGold ignores its input. So just consider *TryGold* running on ε. *TryGold* will accept ε iff there is a number that violates Goldbach's conjecture. So, if *Oracle* exists, the following Turing machine *M* decides *G*: Run *Oracle*(<*TryGold*, ε>). If it accepts, return *False*, else return *True*.

22) A language L is ***D-complete*** iff (1) L is in D, and (2) for every language L' in D, $L' \leq_M L$. Consider the following claim: If $L \in$ D and $L \neq \Sigma^*$ and $L \neq \varnothing$, then L is D-complete. Prove or disprove this claim.

The claim is true. Let L be any language that meets the three requirements of the claim. To show that it is D-complete we must show that it is in D and that every other language in D is mapping reducible to it. We are given that it is in D. So it remains to show that every other decidable language is mapping reducible to it.

Let L' be an arbitrary language in D. We show that it is mapping reducible to L by exhibiting a mapping reduction R.

Let s_1 be any element of L. Since $L \neq \varnothing$, such an element must exist.
Let s_2 be any element of $\neg L$. $L \neq \Sigma^*$, such an element must exist.
Let $M_{L'}$ be a Turing machine that decides L'. Since L' is in D, such a Turing machine must exist.

Define $R(x)$ =
Run $M_{L'}$ on x. If it accepts, then return s_1; else return s_2.

To conclude the proof, we must show that R is correct (i.e., that $x \in L'$ iff $R(x) \in L$). We consider the two cases:

- $x \in L'$: $R(x) = s_1$, which is in L.
- $x \notin L'$: $R(x) = s_2$, which is not in L.

22 Undecidable Languages That Do Not Ask Questions about Turing Machines

1) Solve the linear Diophantine farmer problem presented in Section 22.1.

5 cows; 1 pig; and 94 chickens.

2) Consider the following instance of the Post Correspondence problem. Does it have a solution? If so, show one.

	X	Y
1	a	bab
2	bbb	bb
3	aab	ab
4	b	a

2, 1, 2 is a solution.

3) Prove that, if we consider only PCP instances with a single character alphabet, PCP is decidable.

If there is any index i such that $X_i = Y_i$, then the single element sequence i is a solution.

If there is some index i with the property that $|X_i| > |Y_i|$ and another index j with the property that $|X_j| < |Y_j|$ then there is a solution. In this case, there must be values of n, m, k, and p such that (giving the name a to the single element of Σ):

	X	Y
i:	…. a^{n+k} …..	a^n
j:	a^m	a^{m+p}

The sequence $i^p j^k$ must be a solution since:

- the number of a's in the X string will then be $p(n+k)+km = pn + pk + km$, and
- the number of a's in the Y string will then be $pn + k(m+p) = pn + kp + km$.

For example, suppose that we have:

	X	Y
1:	…. aaaaaaa	aaa
2:	aa	aaaaaaa

We can restate that as:

	X	Y
1:	…. a^{3+4}	a^3
2:	a^2	a^{2+5}
	So: $n = 3$, $k = 4$	$m = 2$, $p = 5$

So 1, 1, 1, 1, 1, 2, 2, 2, 2 is a solution:

X:	…. $a^7a^7a^7a^7a^7a^2a^2a^2a^2 = a^{43}$
Y:	$a^3a^3a^3a^3a^3a^7a^7a^7a^7 = a^{43}$

If, on the other hand, neither of these conditions is satisfied, there is no solution. Now either all the X strings are longer than their corresponding Y strings or vice versa. In either case, as we add indices to any proposed solutions, the lengths of the resulting X and Y strings get farther and farther apart.

4) Prove that, if an instance of the Post Correspondence problem has a solution, it has an infinite number of solutions.

If S is a solution, then S^2, S^3, … are all solutions.

5) Recall that the size of an instance P of the Post Correspondence Problem is the number of strings in its X list. Consider the following claim about the Post Correspondence problem: for any n, if P is a PCP instance of size n and if no string in either its X or its Y list is longer than n, then, if P has any solutions, it has one of length less than or equal to 2^n. Is this claim true or false? Prove your answer.

False. One proof is by the counterexample given in Example 22.3. Another proof is the following. Suppose the claim were true. Then PCP would be decidable. Given any PCP instance P, let *st-length* be the length of the longest string in P's X list or its Y list and let *size* be the size of P. If *st-length* > *size*, then add rows to the X and Y lists so that the two are equal. Create each new row by putting some single character string c_1 in the X list and a different single character string c_2 in the Y list, guaranteeing that neither c_1 nor c_2 occurred in any previous row. Note that these new rows cannot participate in any solution of P, so adding them has no effect on whether or not P has a solution. We now know that *size* ≥ *st-length*, so the claim tells us that, if P has a solution, it has one of length 2^{size} or less. So to find out whether P has a solution, it suffices to try all possible solutions up to length 2^{size}. If a solution is found, accept; else reject. But we know that no decision procedure for PCP exists. So the claim must be false.

6) Let TILES = {<T> : any finite surface on the plane can be tiled, according to the rules described in the book, with the tile set T}. Let s be the string that encodes the following tile set:

Is $s \in$ TILES? Prove your answer.

No. First consider starting with the first tile. The only one that can go to its right is 3. The only one that can go beneath it is 2. But then, next to that 2, we need a tile with black on the left and black on the top. No such tile exists. So tile 1 can participate in no tiling that continues to its right and down. Now suppose that we start with tile 2. Any tile to its right must have a black left side. Neither 2 nor 3 does. So we cannot use 2. Tile 3 cannot tile only with itself. Any tile beneath it must have black on the top, but it doesn't.

7) State whether or not each of the following languages is in D and prove your answer.
   a) {<G> : G is a context-free grammar and $\varepsilon \in L(G)$}.

L is in D. L can be decided by the procedure:
    If *decideCFL*(L, ε) returns *True*, accept. Else reject.

   b) {<G> : G is a context-free grammar and $\{\varepsilon\} = L(G)$}.

L is in D. By the context-free pumping theorem, we know that, given a context-free grammar G, if there is a string of length greater than b^n in $L(G)$, then vy can be pumped out to create a shorter string also in $L(G)$ (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^n. This means that if there are any strings in $L(G)$, there are some strings of length less than b^n. So, to see whether $L(G) = \{\varepsilon\}$, we do the following: First see whether $\varepsilon \in L(G)$ by seeing whether *decideCFL*(L, ε) returns *True*. If not, say no. If ε is in $L(G)$, then we need to determine

whether any other strings are also in $L(G)$. To do that, we test all strings in Σ^* of length up to b^{n+1}. If we find one, we say no, $L(G) \neq \{\varepsilon\}$. If we don't find any, we can assert that $L(G) = \{\varepsilon\}$. Why? If there is a longer string in $L(G)$ and we haven't found it yet, then we know, by the pumping theorem, that we could pump out vy from it until we got a string of length b^n or less. If ε were not in $L(G)$, we could just test up to length b^n and if we didn't find any elements of $L(G)$ 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 $L(G)$, what about the case where we pump out and get ε? That's why we go up to b^{n+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^{n+1} since that's the longest string we can pump out.

c) $\{<G_1, G_2> : G_1 \text{ and } G_2 \text{ are context-free grammars and } L(G_1) \subseteq L(G_2)\}$.

L is not in D. If it were, then we could reduce $GG_= = \{<G_1, G_2> : G_1 \text{ and } G_2 \text{ are CFGs and } L(G_1) = L(G_2)\}$ to it and we have shown that $GG_=$ is not in D . 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,we define R as follows:

$R(<G_1, G_2>) =$
1. If $M_2(<G_1, G_2>)$ accepts and $M_2(<G_2, G_1>)$ accepts then accept, else reject.

If *Oracle* exists and decides L, then $C = R(<G_1, G_2>)$ decides $GG_=$:

- $<G_1, G_2> \in GG_=$: $L(G_1) = L(G_2)$, so $L(G_1) \subseteq L(G_2)$ and $L(G_2) \subseteq L(G_1)$. So M_2 accepts.
- $<G_1, G_2> \notin GG_=$: $L(G_1) \neq L(G_2)$, so $\neg(L(G_1) \subseteq L(G_2)$ and $L(G_2) \subseteq L(G_1))$. So M_2 rejects.

But no machine to decide $GG_=$ can exist, so neither does *Oracle*.

d) $\{<G> : G \text{ is a context-free grammar and } \neg L(G) \text{ is context free}\}$.

e) $\{<G> : G \text{ is a context-free grammar and } L(G) \text{ is regular}\}$.

L is not in D, which we prove by reduction from PCP. The idea is the following. Let P be an instance of PCP. If P has any solution, it has an infinite number of solutions since, if $i_1\ i_2\ \ldots\ i_k$ is a solution, then so is $i_1\ i_2\ \ldots\ i_k i_1\ i_2\ \ldots\ i_k$, $i_1\ i_2\ \ldots\ i_k i_1\ i_2\ \ldots\ i_k i_1\ i_2\ \ldots\ i_k$, and so forth.

As in the text, for any PCP instance P, we can define the following two grammars G_x and G_y:

- $G_x = (\{S_x\} \cup \Sigma \cup \Sigma_n, \Sigma \cup \Sigma_n, R_x, S_x)$, where R_x contains:
  For each value of i between 1 and n: $S_x \rightarrow x_i S_x i$, $S_x \rightarrow x_i i$, where i is represented by the i^{th} element of Σ_n.

- $G_y = (\{S_y\} \cup \Sigma \cup \Sigma_n\}, \Sigma \cup \Sigma_n, R_y, S_y)$, where R_y contains:
  For each value of i between 1 and n: $S_y \rightarrow y_i S_y i$, $S_y \rightarrow y_i i$, where i is represented by the i^{th} element of Σ_n.

Observe first that every string that either G_x or G_y or generates is of the form $x_{i1}x_{i2}\ldots x_{ik}\ \ (i_1, i_2, \ldots i_k)^R$. Since every x_i is an element of Σ^+, the number of characters in the $x_{i1}x_{i2}\ldots x_{ik}$ sequence must be greater than or equal to the number of characters in the $i_1, i_2, \ldots i_k$ sequence. The two sequences are composed of symbols from nonoverlapping alphabets, so the boundary between the two segments of every generated string is clear.

Consider the language $L' = L(G_x) \cap L(G_y)$. We showed that L' is equal to $\varnothing$ iff P has no solution (because there is no string that can be generated, using one sequence of indices, from both the X and the Y lists). $\varnothing$ is regular.

Now suppose that P does have a solution. Then, as we just saw, it has an infinite number of them. So $L' = L(G_x) \cap L(G_y)$ is infinite. And it is not regular since, in every string in it, the length of the $x_{i1}x_{i2}\ldots x_{ik}$ sequence must be at least as long as the length of the $i_1, i_2, \ldots i_k$ sequence. So if we could create a grammar for L' and ask whether that grammar generated a regular language we would be able to determine whether P has a solution.

But there's a problem. Since the context-free languages are not closed under intersection, we cannot be guaranteed to be able to construct a grammar for L'. But we observe that:

$$L' = L(G_x) \cap L(G_y) = \neg(\neg L(G_x) \cup \neg L(G_y)).$$

Both $L(G_x)$ and $L(G_2)$ are deterministic context-free, so their complements are context-free. Thus the union of the complements is also context-free. The only thing we can't do is to take the complement of that. So let's change the language we work with.

Let $L'' = \neg(L(G_x) \cap L(G_y)) = \neg L(G_x) \cup \neg L(G_y)$. L'' is context-free and we can build a grammar for it using the algorithms presented in the book. If P has no solutions, then $L(G_x) \cap L(G_y) = \varnothing$. So $L'' = \neg(L(G_x) \cap L(G_y)) = \Sigma^*$, which is regular. If P has any solutions, then L'' is not regular. If it were, then its complement would also be regular (since the regular languages are closed under complement). So $L' = L(G_x) \cap L(G_y)$ would be regular. But then $(L')^R$ would also be regular because the regular languages are closed under reverse. Every string in $(L')^R$ has a first part that is a sequence of indices and a second part that is a corresponding sequence of strings from the X list, and the length of the first part must be less than or equal to the length of the second part (since each index must produce at least one character). We can use the Pumping Theorem for regular languages to show that $(L')^R$ is not regular. Let w be a shortest string in $(L')^R$ of the form $i^n c^m$, such that $n > k$ (from the Pumping Theorem), where i is an index and c is a character in Σ. y must fall in the index region, so it is some nonempty sequence of index characters. Let q be m+1. The resulting string has n+m index characters and only m characters from the X list. Since m is not 0, n+$m > n$, so the resulting string is not in $(L')^R$.

We are now ready to state a reduction $(R, \neg)$ from PCP to L. Define R as follows:

$R(\langle P \rangle) =$

1. From P construct G_x and G_y as described above.
2. From them, construct a new grammar G that generates $\neg L(G_x) \cup \neg L(G_y)$.
3. Return $\langle G \rangle$.

If *Oracle* exists and decides L, then $C = \neg M_2(R(\langle P \rangle))$ decides PCP:

- $\langle P \rangle \in$ PCP: P has some solution so $\neg L(G_x) \cup \neg L(G_y)$ is not empty and, by the argument we just gave, is not regular. *Oracle* rejects, so C accepts.
- $\langle P \rangle \notin$ PCP: If P has no solution, then $\neg L(G_x) \cup \neg L(G_y) = \Sigma^*$, which is regular. *Oracle* accepts, so C rejects.

But no machine to decide PCP can exist, so neither does *Oracle*.

23 Unrestricted Grammars

1) Write an unrestricted grammar for each of the following languages L:

a) $\{a^{2^n}\ b^{2^n}, n \geq 0\}$.

$S \rightarrow$ # ab %	
# $\rightarrow$ # D	/* Each D is a doubler. Spawn (n-1) of them. Each of them will get pushed to the right and will turn each a into aa and each b into bb as it passes over.
Da $\rightarrow$ aa$D$	/* Move right and double.
Db $\rightarrow$ bb$D$	″
D% $\rightarrow$ %	/* D's work is done. Wipe it out.
# $\rightarrow$ ε	/* Get rid of the walls.
% $\rightarrow$ ε	″

G generates all strings in L: If no D's are generated, G generates ab (n = 0). For any other value of n, the correct number of D's can be generated. G generates only strings in L: Once a D is generated, it cannot be eliminated until it has made it all the way to the right and is next to %. To do that, it must double each a and each b it passes over.

b) $\{a^n b^m c^{n+m} : n, m > 0\}$.

$S \rightarrow$ a S c	/* L is actually context free.
$S \rightarrow$ a T c	
$T \rightarrow$ b T c	
$T \rightarrow$ b c	

c) $\{a^n b^m c^{nm} : n, m > 0\}$.

$S \rightarrow S_1$ #	/* First generate $A^n b^m$#
$S_1 \rightarrow A\ S_1$	
$S_1 \rightarrow A\ S_2$	
$S_2 \rightarrow S_2$ b	
$S_2 \rightarrow$ b	
$A \rightarrow$ a 1	/* For each A, in order to convert it to a, we will generate a 1. /* Then we'll push the 1 rightwards. As it passes over the b's, it will generate a C for each b. Start with the rightmost A or the second 1 will get stuck.
1 a $\rightarrow$ a 1	
1 b $\rightarrow$ b C 1	
C b $\rightarrow$ b C	/* Move all the C's to the right of the b's.
C 1 # $\rightarrow$ 1 # c	/* Jump each C across # and convert it to c.
1 # $\rightarrow$ #	/* Get rid of 1 once all the C's have jumped. If it goes too soon, then some C's will be stuck to the left of #.
b# c $\rightarrow$ bc	/* Get rid of # at the end.

d) $\{a^n b^{2n} c^{3n} : n \geq 1\}$.

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

$Ba \rightarrow aB$
$Bc \rightarrow bbc$
$Bb \rightarrow bbb$

e) $\{ww^Rw : w \in \{a, b\}^*\}$.

$S \rightarrow S_1 \#$	
$S_1 \rightarrow a\ S_1\ a$	/* First generate $wTw^R\#$
$S_1 \rightarrow b\ S_1\ b$	
$S_1 \rightarrow T$	
	/* Take each character of w^R, starting at the left. Make a copy of it and slide it to the right of the # to create the second w. Use 1 for a and 2 for b.
$T\ a \rightarrow T\ 1\ A$	/* We'll move A for a, B for b. And we'll transform each character of w^R as it is considered: a will be 1, b will be 2. These two rules will handle the first such character.
$T\ b \rightarrow T\ 2\ B$	
$1\ a \rightarrow 1\ 1\ A$	/* These next four will do this for characters 2 through n of w^R.
$1\ b \rightarrow 1\ 2\ B$	
$2\ a \rightarrow 2\ 1\ A$	
$2\ b \rightarrow 2\ 2\ B$	
$A\ a \rightarrow a\ A$	/* Push A's and B's to the right until they hit #.
$A\ b \rightarrow b\ A$	
$B\ b \rightarrow b\ B$	
$B\ a \rightarrow a\ B$	
$A\ \# \rightarrow \#\ a$	/* Jump across #
$B\ \# \rightarrow \#\ b$	
$1\ \# \rightarrow \#\ a$	/* Once all of w^R has been converted to 1's and 2's (i.e., it's all been copied), push # leftward converting 1's back to a's and 2's to b's.
$2\ \# \rightarrow \#\ b$	
$T\ \# \rightarrow \varepsilon$	/* Done.

f) $\{a^n b^n a^n b^n : n \geq 0\}$.

$S \rightarrow \%\ T$	
$T \rightarrow A\ B\ T\ A\ B$	/* Generate n A's and n B's on each side of the middle marker #.
$T \rightarrow \#$	/* At this point, strings will look like: $\%ABABAB\#ABABAB$.
$BA \rightarrow AB$	/* Move B's to the right of A's.
$\%A \rightarrow a$	/* Convert A's to a's and B's to b's, moving left to right.
$aA \rightarrow aa$	"
$aB \rightarrow ab$	
$bB \rightarrow bb$	
$\#A \rightarrow a$	
$\% \# \rightarrow \varepsilon$	/* The special case where no A's and B's are generated.

g) $\{xy\#x^R : x, y \in \{a, b\}^*$ and $|x| = |y|\}$.

$S \rightarrow a\ S\ X\ a$	/* Generate $x\#x^R$, with X's mixed in. The X's will become y.
$S \rightarrow b\ S\ X\ b$	
$S \rightarrow \#$	/* At this point, we have something like aab#XbXaXa.
$b\ X \rightarrow X\ b$	/* Push all the X's to the left, up next to #.
$a\ X \rightarrow X\ a$	/* At this point, we have something like aab#XXXbaa.

$\# X \rightarrow \texttt{a} \#$	/* Hop each X leftward over # and convert it to a or b.
$\# X \rightarrow \texttt{b} \#$	

h) $\{w\texttt{c}^m\texttt{d}^n : w \in \{\texttt{a}, \texttt{b}\}^*$ and $m = \#_{\texttt{a}}(w)$ and $n = \#_{\texttt{b}}(w)\}$.

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:

$S \rightarrow S_1\#$
$S_1 \rightarrow \texttt{a}S_1C$
$S_1 \rightarrow \texttt{b}S_1D$
$S_1 \rightarrow \varepsilon$
$DC \rightarrow CD$
$D\# \rightarrow \texttt{d}$
$D\texttt{d} \rightarrow \texttt{dd}$
$C\# \rightarrow \texttt{c}$
$C\texttt{d} \rightarrow \texttt{cd}$
$C\texttt{c} \rightarrow \texttt{cc}$
$\# \rightarrow \varepsilon$

So with R as given above, the grammar $G = (\{S, S_1, C, D, \#, \texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}, \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}, R, S\}$.

2) Show a grammar that computes each of the following functions (given the input convention described in Section 23.3).

a) f: $\{\texttt{a}, \texttt{b}\}^+ \rightarrow \{\texttt{a}, \texttt{b}\}^+$, where $f(s = \texttt{a}_1\texttt{a}_2\texttt{a}_3\ldots\texttt{a}_{|s|}) = \texttt{a}_2\texttt{a}_3\ldots\texttt{a}_{|s|}\texttt{a}_1$. For example $f($aabbaa$)$ = abbaaa.

So what we need is a a grammar that converts, for example Saabbaa$S$ into abbaaa.

$S\texttt{a} \rightarrow A$	/* Remember that we need to add an a to the right hand end.
$S\texttt{b} \rightarrow B$	/* Remember that we need to add a b to the right hand end.
$A\texttt{a} \rightarrow \texttt{a}A$	/* Push the A or B all the way to the right.
$A\texttt{b} \rightarrow \texttt{b}A$	/* "
$B\texttt{a} \rightarrow \texttt{a}B$	/* "
$B\texttt{b} \rightarrow \texttt{b}B$	/* "
$AS \rightarrow \texttt{a}$	/* Tack on the final a or b.
$BS \rightarrow \texttt{b}$	/* "

b) f: $\{\texttt{a}, \texttt{b}\}^+ \rightarrow \{\texttt{a}, \texttt{b}, \texttt{1}\}^+$, where $f(s) = s1^n$, where $n = \#_{\texttt{a}}(s)$. For example $f($aabbaa$)$ = aabbaa1111.

$S\texttt{a} \rightarrow \texttt{a}SQ$	/* Generate a Q for every a.
$S\texttt{b} \rightarrow \texttt{b}S$	/* Skip over b's.
$Q\texttt{a} \rightarrow \texttt{a}Q$	/* Push all the Q's to the right.
$Q\texttt{b} \rightarrow \texttt{b}Q$	/* "
$QS \rightarrow S\texttt{1}$	/* Convert each Q to a 1 and hop it over the right hand S.
$SS \rightarrow \varepsilon$	

c) f: {a, b}*#{a, b}* → {a, b}*, where $f(x\#y) = xy^R$.

$G = (\{S, P, W, \mathtt{a}, \mathtt{b}, \#\}, \{\mathtt{a}, \mathtt{b}, \#\}, R, S\}$, where R =

$S\,\mathtt{a} \rightarrow \mathtt{a}$	/* Get rid of initial S.
$S\,\mathtt{b} \rightarrow \mathtt{b}$	
$\# \rightarrow \#\,P$	/* Generate a pusher. We will push the characters in y to the right, starting
$P\,\mathtt{a}\,\mathtt{a} \rightarrow \mathtt{a}\,P\,\mathtt{a}$	with the leftmost one.
$P\,\mathtt{a}\,\mathtt{b} \rightarrow \mathtt{b}\,P\,\mathtt{a}$	
$P\,\mathtt{b}\,\mathtt{a} \rightarrow \mathtt{a}\,P\,\mathtt{b}$	
$P\,\mathtt{b}\,\mathtt{b} \rightarrow \mathtt{b}\,P\,\mathtt{b}$	
$P\,\mathtt{a}\,S \rightarrow W\,\mathtt{a}$	/* Hop the first character over the right wall and change the wall from S to W
$P\,\mathtt{b}\,S \rightarrow W\,\mathtt{b}$	so that the wall cannot get wiped out by rules 1 and 2.
$P\,\mathtt{a}\,W \rightarrow W\,\mathtt{a}$	/* Hop the other characters over the wall once they get all the way rightward.
$P\,\mathtt{b}\,W \rightarrow W\,\mathtt{b}$	
$\#\,W \rightarrow \varepsilon$	/* When all characters in y have hopped, wipe out # W and we're done.

d) f: {a, b}$^+$ → {a, b}$^+$, where $f(s)$ = if $\#_{\mathtt{a}}(s)$ is even then s, else s^R.

We first have to determine whether the number of a's is even or odd. To do that, create the symbol C at the right end of the string. Push it leftward. If it crosses an a, change it to a D. Keep pushing. If D crosses an a, change it back to C. If the symbol that runs into the left hand S is a C, the a count is even. If it's a D, the a count is odd.

$\mathtt{a}S \rightarrow \mathtt{a}CT$	
$\mathtt{b}S \rightarrow \mathtt{b}CT$	
$\mathtt{a}C \rightarrow D\mathtt{a}$	
$\mathtt{b}C \rightarrow C\mathtt{b}$	
$\mathtt{a}D \rightarrow C\mathtt{a}$	
$\mathtt{b}D \rightarrow D\mathtt{b}$	
$SC \rightarrow X$	/* Count of a's is even. All done except to delete final T.
$X\mathtt{a} \rightarrow \mathtt{a}X$	"
$X\mathtt{b} \rightarrow \mathtt{b}X$	"
$XT \rightarrow \varepsilon$	"
$SD \rightarrow SD\#$	/* Count of a's is odd. Must reverse. # will be a pusher.
$\#\mathtt{a}\mathtt{a} \rightarrow \mathtt{a}\#\mathtt{a}$	"
$\#\mathtt{a}\mathtt{b} \rightarrow \mathtt{b}\#\mathtt{a}$	"
$\#\mathtt{b}\mathtt{a} \rightarrow \mathtt{a}\#\mathtt{b}$	"
$\#\mathtt{b}\mathtt{b} \rightarrow \mathtt{b}\#\mathtt{b}$	"
$\#\mathtt{b}T \rightarrow T\mathtt{b}$	"
$\#\mathtt{a}T \rightarrow T\mathtt{a}$	"
$SDT \rightarrow \varepsilon$	"

e) f: {a, b}* → {a, b}*, where $f(w) = ww$.

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 → 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 → bS#b	Same for copying b.
#aa → a#a	Move the a we're copying past the next character if it's an a.
#ab → b#a	Move the a we're copying past the next character if it's a b.
#ba → a#b	Same two rules for pushing b.
#bb → b#b	"
#aS → #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 → #aTW	Same if we get to the end of w pushing b.
#aT → 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 → 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.
$ST \rightarrow \varepsilon$	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.
$W \rightarrow \varepsilon$	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}

f) f: {a, b}$^+$ → {a, b}*, where $f(s)$ = if $|s|$ is even then s, else s with the middle character chopped out. (Hint: the answer to this one is fairly long, but it is not very complex. Think about how you would use a Turing machine to solve this problem.)

Here's one approach. This one follows very closely the way we would solve this problem with a Turing machine. See below for a simpler solution that takes better advantage of the grammar formalism. In both cases, the key is to find the middle of the string.

The basic idea is that we'll introduce a marker Q that will start at the left of the string. (We make sure only to do this once by doing it when the first character after the initial S is a or b. It will never be that again once we start marking off.) The Q will mark off one character (we'll use 1 for a and 2 for b). It will then move all the way across the string. When it gets to the right, it will mark off a character and turn into P, whose job is to move back leftward. When it gets to the last unmarked character, it will mark it and change back to Q, ready for another pass to the right. And so forth. Eventually one of two things will happen: The two marked regions will exactly come together in the middle (in case the string is of even length) or there will be one extra character (in case it's of odd length). If odd, we'll wipe out the extra character (which will be the last one we converted as we started our last pass rightward). In either case, we'll generate TT in the middle of the string. The left hand T will move leftward, flipping 1's back to a's and 2 back to b's. The right hand T will move rightward doing the same thing. When either T hits the S on the end, the two disappear.

SaS → ε	/* Treat one character inputs as a special case
SbS → ε	
Sa → S1Q	/* Introduce Q and flip the first character at the same time.
Sb → S2Q	
Qa → aQ	/* Push Q to the right.
Qb → bQ	
aQS → P1S	/* The first time we move right, we'll hit the S at the end.
bQS → P2S	/* Flip the character immediately before the S and flip Q to P.
aQ1 → P11	/* All but the first time we go right, we'll hit a flipped character
bQ1 → P21	rather than the S. But we do the same thing. Flip the
aQ2 → P12	next character and flip Q to P.
bQ2 → P22	
aP → Pa	/* Push P to the left.
bP → Pb	
1Pa → 11Q	/* We've hit the left (the characters we have already flipped). So
1Pb → 12Q	so flip the next one and flip P back to Q ready to go right again.
2Pa → 21Q	
2Pb → 22Q	
1P1 → 1TT1	/* We pushed P all the way left but there are no a's or b's remaining
1P2 → 1TT2	on its right so we've found the middle. We marked off the same
2P1 → 2TT1	number of characters in both directions so string was even.
2P2 → 2TT2	So keep all characters and introduce TT.

```
1Q1 → TT1       /* We just flipped a character and created Q.  But all the characters to
1Q2 → TT2               its right are already flipped.  So there's no mate to the last char
2Q1 → TT1               we flipped.  String is odd.  Erase last flipped character (to the
2Q2 → TT2               left of Q and introduce TT.

1T → Ta         /* Moving the left hand T leftward.  Each character it goes across, it
2T → Tb                 flips.
T1 → aT         /* Moving the right hand T rightward.  Each character it goes across,
T2 → bT                  it flips.

ST → ε          /* Done on the right.
TS → ε          /* Done on the left.
```

Here's an alternative solution that doesn't require marking off the a's and b's and then putting them back when we're done. The basic idea is that at each pass, instead of marking off one character on the left and one on the right, we'll jump them to the other side of the end marker. We'll see whether there's a character left in the middle when the two end markers come together.

```
SaS → ε         /* Treat one-character inputs as a special case.
SbS → ε                 ""

Sa → aTQ        /* Jump the first character over the initial S and create Q, which we'll
Sb → bTQ                push all the way to the right to jump a corresponding character.

Qa → aQ         /* Push Q all the way to the right.
Qb → bQ                         ""

aQS → WSa       /* Jump the last character over the righthand S and convert Q to
bQS → WSb               W for the trip back leftward.  We've now jumped one
                        character on the left and one on the right.

aW → Wa         /* Push W all the way to the left.
bW → Wb                 ""

TWa → aTQ       /* When W makes it all the way back to the left, grab the next
TWb → bTQ               character and jump it over the end marker (T).  Then convert W
                        to Q for the next trip rightward.

                /* After all pairs have been jumped across the outside walls, we'll
                        be left in the middle with either TWaS or TWbS or TWS; the former
                        in case there were an odd number of characters, the second in case
                        there were an even number.  In the former case, the rules we have
                        now might jump the remaining character over, but if they do, they
                        will also change the W to Q.  We will make sure here that the only
                        way to get rid of the T and the S is still to have the leftward
                        moving W around, so that branch will die.
TWaS → ε        /* Nuke the middle character and the delimiters and we're done.
TWbS → ε                ""

TWS → ε         /* Nuke the delimiters and we're done. The string was even.
```

g) $f(n) = m$, where $value_1(n)$ is a natural number and $value_1(m) = value_1(\lfloor n/2 \rfloor)$. Recall that $\lfloor x \rfloor$ (read as "floor of x") is the largest integer that is less than or equal to x.

h) $f(n) = m$, where $value_2(n)$ is a natural number and $value_2(m) = value_2(n) + 5$.

3) Show that, if G, G_1, and G_2 are unrestricted grammars, then each of the following languages, defined in Section 23.4, is not in D:

a) $L_b = \{<G> : \varepsilon \in L(G)\}$.

We show that $A_\varepsilon \leq_M L_b$ and so L_b is not decidable. Let R be a mapping reduction from $A_\varepsilon = \{<M> :$ Turing machine M accepts $\varepsilon\}$ to L_b, defined as follows:

$R(<M>) =$
1. From M, construct the description $<G\#>$ of a grammar $G\#$ such that $L(G\#) = L(M)$.
2. Return $<G\#>$.

If *Oracle* exists and decides L_b, then $C = Oracle(R(<M>))$ decides A_ε. R can be implemented as a Turing machine using the algorithm presented in Section 23.2. And C is correct:

- If $<M> \in A_\varepsilon$: $M(\varepsilon)$ halts and accepts. $\varepsilon \in L(M)$. So $\varepsilon \in L(G\#)$. $Oracle(<G\#>)$ accepts.
- If $<M> \notin A_\varepsilon$: $M(\varepsilon)$ does not halt. $\varepsilon \notin L(M)$. So $\varepsilon \notin L(G\#)$. $Oracle(<G\#>)$ rejects.

But no machine to decide A_ε can exist, so neither does *Oracle*.

b) $L_c = \{<G_1, G_2> : L(G_1) = L(G_2)\}$.

We show that EqTMs $\leq_M L_c$ and so L_c is not decidable. Let R be a mapping reduction from EqTMs = EqTMs $= \{<M_a, M_b> : L(M_a) = L(M_b)\}$ to L_c, defined as follows:

$R(<M_a, M_b>) =$
1. From M_a, construct the description $<G_a\#>$ of a grammar $G_a\#$ such that $L(G_a\#) = L(M_a)$.
2. From M_b, construct the description $<G_b\#>$ of a grammar $G_b\#$ such that $L(G_b\#) = L(M_b)$.
3. Return $<G_a\#, G_b\#>$.

If *Oracle* exists and decides L_c, then $C = Oracle(R(<M_a, M_b>))$ decides EqTMs. R can be implemented as a Turing machine using the algorithm presented in Section 23.2. And C is correct:

- If $<M_a, M_b> \in$ EqTMs : $L(M_a) = L(M_b)$. $L(G_a\#) = L(G_b\#)$. $Oracle(<G_a\#, G_b\#>)$ accepts.
- If $<M_a, M_b> \notin$ EqTMs : $L(M_a) \neq L(M_b)$. $L(G_a\#) \neq L(G_b\#)$. $Oracle(<G_a\#, G_b\#>)$ rejects.

But no machine to decide EqTMs can exist, so neither does *Oracle*.

c) $L_d = \{<G> : L(G) = \varnothing\}$.

The proof is by reduction from $A_{ANY} = \{<M> :$ there exists at least one string that Turing machine M accepts$\}$. Define:

$R(<M>) =$
1. From M, construct the description $<G\#>$ of a grammar $G\#$ such that $L(G\#) = L(M)$.
2. Return $<G\#>$.

$\{R, \neg\}$ is a reduction from A_{ANY} to L_d. If *Oracle* exists and decides L_d, then $C = \neg Oracle(R(<M>))$ decides A_{ANY}. R can be implemented as a Turing machine using the algorithm presented in Section 23.2. And C is correct:

- If $<M> \in A_{ANY}$: M accepts at least one string. $L(M) \neq \varnothing$. So $L(G\#) \neq \varnothing$. *Oracle*(<$G\#$>) rejects. C accepts.
- If $<M> \notin A_{ANY}$: M does not accept at least one string. $L(M) = \varnothing$. So $L(G\#) = \varnothing$. *Oracle*(<$G\#$>) accepts. C rejects.

But no machine to decide $A_{ANY\varepsilon}$ can exist, so neither does *Oracle*.

4) Show that, if G is an unrestricted grammar, then each of the following languages is not in D:
   a) {<G> : G is an unrestricted grammar and $\mathtt{a}^* \subseteq L(G)$}.

Let R be a mapping reduction from H to L defined as follows:

$R(<M, w>)$ =

1. Construct the description <$M\#$> of a new Turing machine $M\#(x)$, which operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Build the description <$G\#$> of a grammar $G\#$ such that $L(G\#) = L(M\#)$.
3. Return <$G\#$>.

If *Oracle* exists, then $C = Oracle(R(<M, w>))$ decides L. R can be implemented as a Turing machine. And C is correct. $G\#$ generates Σ^* or $\varnothing$, depending on whether M halts on w. So:

- $< M, w> \in H$: M halts on w, so $M\#$ accepts all inputs. $G\#$ generates Σ^*. $\mathtt{a}^* \subseteq \Sigma^*$. *Oracle* accepts.
- $< M, w> \notin H$: M does not halt on w, so $M\#$ halts on nothing. $G\#$ generates $\varnothing$. It is not true that $\mathtt{a}^* \subseteq \varnothing$. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

   b) {<G> : G is an unrestricted grammar and G is ambiguous}. Hint: Prove this by reduction from PCP.

5) Let G be the unrestricted grammar for the language $A^nB^nC^n = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$, shown in Example 23.1. Consider the proof, given in Section 36.4, of the undecidability of the Post Correspondence Problem. The proof is by reduction from the membership problem for unrestricted grammars.
   a) Define the MPCP instance MP that will be produced, given the input <G, abc>, by the reduction that is defined in the proof of Theorem 36.1.

	X	Y
1	%S ⇒	%
2	&	⇒ abc%
3	S	S
4	B	B
5	a	a
6	b	b
7	c	c
8	aBSc	S
9	ε	S
10	aB	Ba
11	bc	Bc
12	bb	Bb
13	⇒	⇒

   b) Find a solution for MP.

1, 8, 13, 5, 4, 9, 7, 13, 5, 11, 2.

c) Define the PCP instance P that will be built from MP by the reduction that is defined in the proof of Theorem 36.2.

	A	B
0	¢%¢S¢ ⇒¢	¢%
1	%¢S¢ ⇒¢	¢%
2	&¢	¢⇒ ¢a¢b¢c¢%
3	S¢	¢S
4	B¢	¢B
5	a¢	¢a
6	b¢	¢b
7	c¢	¢c
8	a¢B¢S¢c¢	¢S
9	ε	¢S
10	a¢B¢	¢B¢a
11	b¢c¢	¢B¢c
12	b¢b¢	¢B¢b
13	⇒¢	¢⇒
14	$	¢$

d) Find a solution for P.

0, 8, 13, 5, 4, 9, 7, 13, 5, 11, 2, 14.

24 Context-Sensitive Languages and the Chomsky Hierarchy

1) Write context-sensitive grammars for each of the following languages L. The challenge is that, unlike with an unrestricted grammar, it is not possible to erase working symbols.
   a) $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$.
   b) $WW = \{ww : w \in \{a, b\}^*\}$.
   c) $\{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}$.

2) Prove that each of the following languages is context-sensitive:
   a) $\{a^n : n \text{ is prime}\}$.
   b) $\{a^{n^2} : n \geq 0\}$.
   c) $\{xwx^R : x, w \in \{a, b\}^+ \text{ and } |x| = |w|\}$.

Describe LBAs.

3) Prove that every context-free language is accepted by some deterministic LBA.

Every context-free language L can be accepted by some (possibly nondeterministic) PDA P. Every PDA can be simulated by some (possibly nondeterministic) two-tape Turing machine M that uses its second tape to store the PDA's stack. Examine P and determine the largest number of symbols that can be pushed onto the stack in a single move. Call that number s. On input w, the height of P's stack is never longer than $s \cdot |w|$. So, with an appropriate encoding for its two tapes, M will never use more tape squares than $|w|$. Every nondeterministic Turing machine can be simulated by a deterministic one and the length of the tape that is required to do so is $|w|$. So there is a deterministic Turing machine M that accepts L. And since the number of tape squares M uses is $|w|$, M is an LBA. So M is a deterministic LBA.

4) Recall the diagonalization proof that we used in the proof of Theorem 24.4, which tells us that the context-sensitive languages are a proper subset of D. Why cannot that same proof technique be used to show that there exists a decidable language that is not decidable or an SD language that is not decidable?

An attempt to use this technique to prove that there exists a decidable language that is not decidable (a statement that must be false) fails at the step at which it is required to create a lexicographic enumeration of the decidable languages. No such enumeration exists. It isn't possible to construct one using TMs, since, to do that, it would be necessary to be able to examine a TM and decide whether it is a deciding machine. That can't be done. Similarly, it can't be done using unrestricted grammars.

Now suppose that we try to use this technique to prove that there exists a semidecidable language that is not decidable (a statement that can be proven to be true by exhibiting the language H). The diagonalization technique fails because there exists no procedure that can be guaranteed to fill in values of the table as they are needed.

5) Prove that the context-sensitive languages are closed under *reverse*.

If L is a context-sensitive language, then there is some LBA M that decides it. From M, we construct a new LBA M^* that decides L^R. M^* will treat its tape as though it were divided into two tracks. It begins by copying its input w onto the second track. Next it erases the first track and places the read/write head of the second track on the rightmost character. Then it moves that head leftward while moving the head on track one rightward, copying each character from track one to track two. When this process is complete, track one holds w^R. So M^* then simply runs M. M^* thus accepts exactly the strings in L^R.

6) Prove that each of the following questions is undecidable:
   a) Given a context-sensitive language L, is $L = \Sigma^*$?

b) Given a context-sensitive language L, is L finite?

By reduction from {<M> : M halts on a finite number of strings}. Use reduction via computation history. If M halts on a finite number of strings, then it will have a finite number of computation histories.

c) Given two context-sensitive languages L_1 and L_2, is $L_1 = L_2$?

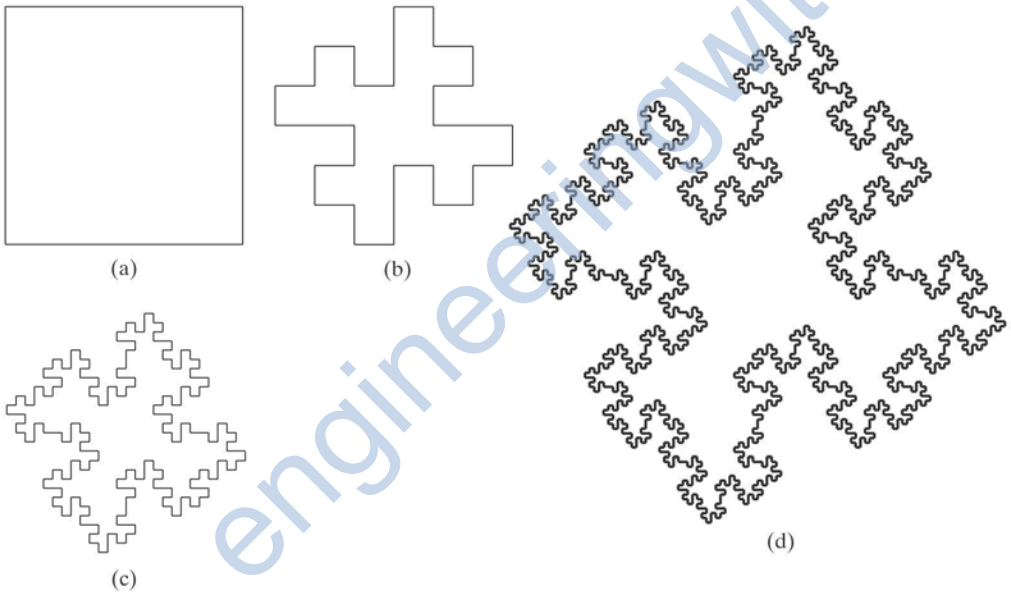
d) Given two context-sensitive languages L_1 and L_2, is $L_1 \subseteq L_2$?

e) Given a context-sensitive language L, is L regular?

7) Prove the following claim, made in Section 24.3: Given an attribute/feature/unification grammar formalism that requires that both the number of features and the number of values for each feature must be finite and a grammar G in that formalism, there exists an equivalent context-free grammar G'.

If the number of features and the number of feature values are both finite, then the number of feature/value combinations is also finite. So we can construct a new grammar that uses one nonterminal symbol for each combination of an original nontermal symbol and a matrix of feature/value combinations.

8) The following sequence of figures corresponds to a fractal called a ***Koch island***:



These figures were drawn by interpreting strings as turtle programs, just as we did in Example 24.5 and Example 24.6. The strings were generated by an L-system G, defined with:

$\Sigma = \{\mathtt{F}, +, -\}$.
$\omega = \mathtt{F} - \mathtt{F} - \mathtt{F} - \mathtt{F}$.

To interpret the strings as turtle programs, attach meanings to the symbols in Σ as follows (assuming that some value for k has been chosen):

- F means move forward, drawing a line of length k.
- + means turn left 90°.
- – means turn right 90°.

Figure (a) was drawn by the first generation string ω. Figure (b) was drawn by the second generation string, and so forth. R_G contains a single rule. What is it?

```
F → F − F + F + FF − F −F + F
```

25 Computable and Partially Computable Functions

1) Define the function *pred*(*x*) as follows:

pred: $\mathbb{N} \rightarrow \mathbb{N}$,
pred(*x*) = *x* - 1.

a) Is *pred* a total function on $\mathbb{N}$?

No, because it is not defined on 0.

b) If not, is it a total function on some smaller, decidable domain?

Yes, it is a total function on the positive integers.

c) Show that *pred* is computable by defining an encoding of the elements of $\mathbb{N}$ as strings over some alphabet Σ and then showing a Turing machine that halts on all inputs and that computes either *pred* or *pred′* (using the notion of a primed function as described in Section 25.1.2).

Encode each natural number *n* as *n* `1`'s. So 0 is encoded as ε. Then there is a Turing machine that computes *pred′*:

M(*x*) =
1. If *x* = ε, output *Error*.
2. Else erase the first `1` of *x* and halt.

2) Prove that every computable function is also partially computable.

If *f* is a computable function, then there is a Turing machine that computes it. Since there is a Turing machine that computes it, it is partially computable.

3) Consider *f*: $A \rightarrow \mathbb{N}$, where $A \subseteq \mathbb{N}$. Prove that, if *f* is partially computable, then *A* is semidecidable (i.e., Turing enumerable).

An algorithm to enumerate *A* is the following: Lexicographically enumerate the elements of $\mathbb{N}$. Use the dovetailing technique to apply *f*: to each of the enumerated elements in an interleaved fashion. Whenever *f*: halts and returns a value for some input *x* then *x* is an element of *A*. Output it.

4) Give an example, other than *steps*, of a function that is partially computable but not computable.

Define *output*(<*M*>) = the number of nonblank squares on *M*'s tape when it halts when started on an empty tape. Output is defined only on those values of <*M*> such that *M* halts on ε. But that's the language H_ε, which we have shown is not in D.

5) Define the function *countL*(<*M*>) as follows:

countL: {<*M*> : *M* is a Turing machine} $\rightarrow \mathbb{N} \cup \{\aleph_0\}$,
countL(<*M*>) = the number of input strings that are accepted by Turing machine *M*.

a) Is *countL* a total function on {<*M*> : *M* is a Turing machine}?

Yes. If we had defined it to map just to ℕ, then it wouldn't be since, if *M* accepts an infinite number of strings, then there is no integer that corresponds to the number of strings it accepts.

b) If not, is it a total function on some smaller, decidable domain?

Not applicable, since it is.

c) Is *countL* computable, partially computable, or neither? Prove your answer.

The function *countL* is not even partially computable. If it were, there would be some Turing machine M_C that partially computes it. Since *countL* is a total function, M_C halts on all inputs. And it must return the value $\aleph_0$ iff $L(M)$ is infinite. So we would be able to use M_C to decide whether $L(M)$ is infinite. But we know we can't do that. Formally, if M_C exists, then the following procedure decides L_{INF} = {<*M*> : $L(M)$ is infinte}:

decideL_{INF}(<*M*>) =
1. Run M_C(<*M*>).
2. If the result is $\aleph_0$, accept. Else reject.

In Exercise 21.1 (t), we showed that *decide*L_{INF} is not even semidecidable, much less decidable. So M_C does not exist.

6) Give an example, other than any mentioned in the book, of a function that is not partially computable.

Define *easiest*(<*M*>) = if there is at least one input on which *M* halts then the smallest number of steps executed by *M* on any input, before it halts
otherwise, -1

Easiest is defined on all inputs <*M*>. If it were partially computable, therefore, it would also be computable. If it were computable, we could use it to find out whether *M* halts on any strings at all. But we know that the language H_{ANY} is not in D.

7) Let *g* be some partially computable function that is not computable. Let *h* be some computable function and let $f(x) = g(h(x))$. Is it possible that *f* is a computable function?

Yes. Let *g* be *steps*(<*M*, *w*>). It's partially computable but not computable. Let *h*(<*M*, *w*>) be the constant function that, on any input, returns <*M′*, ε>, where *M′* is a simple Turing machine that immediately halts. Then *steps*(<*M′*, ε>) = 0. So *f* is the simple computable function that, on all inputs, returns 0.

8) Prove that the busy beaver function Σ is not computable.

Suppose that Σ were computable. Then there would be some Turing machine *BB*, with some number of states that we can call *b*, that computes it. For any positive integer *n*, we can define a Turing machine $Write_n$ that writes *n* 1's on its tape, one at a time, moving rightwards, and then halts with its read/write head on the blank square immediately to the right of the rightmost 1. $Write_n$ has *n* nonhalting states plus one halting state. We can also define a Turing machine *Multiply*, which multiplies two unary numbers, written on its tape and separated by the character #. Call the number of states in *Multiply m*.

Using the macro notation we described in Section 17.1.5, we can now define, for any positive integer *n*, the following Turing machine, which we can call $Trouble_n$:

> $Write_n$ # R $Write_n$ L_θ *Multiply* L_θ *BB*

The number of states in $Trouble_n$ is $2n + m + b + 8$. *BB*, the final step of $Trouble_n$, writes a string of length $\Sigma(n^2)$. Since, for any $n > 0$, $Trouble_n$ is a Turing machine with $2n + m + b + 8$ states that write $\Sigma(n^2)$ 1's, we know that:

$$\Sigma(2n + m + b + 8) \geq \Sigma(n^2)$$

By Theorem 25.3, we know that Σ is monotonically increasing, so it must then also be true that, for any $n > 0$:

$$2n + m + b + 8 \geq n^2$$

But, since n^2 grows faster than n does, that cannot be true. In assuming that *BB* exists, we have derived a contradiction. So *BB* does not exist. So Σ is not computable.

9) Prove that each of the following functions is primitive recursive:

a) The unary function $double(x) = 2x$.

$$double(0) = 0$$
$$double(n+1) = succ(succ(double(n)))$$

b) The proper subtraction function *monus*, which is defined as follows:

$$monus(n, m) = \begin{cases} n - m & \text{if } n > m \\ 0 & \text{if } n \leq m \end{cases}$$

$$monus(n, 0) = n$$
$$monus(n, m+1) = pred(monus(n, m))$$

c) The function *half*, which is defined as follows:

$$half(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ (n - 1)/2 & \text{if } n \text{ is odd} \end{cases}$$

The solution to this can be found on p. 71 of Yasuhara, Ann, Recursive Function Theory & Logic. Academic Press, 1971.

10) Let A be Ackermann's function. Verify that $A(4, 1) = 65533$.

26 Summary of Part IV

Part V: Complexity

27 Introduction to the Analysis of Complexity

1) Let M be an arbitrary Turing machine.
   a) Suppose that $timereq(M) = 3n^3(n+5)(n-4)$. Circle all of the following statements that are true:
      i) $timereq(M) \in \mathcal{O}(n)$. False
      ii) $timereq(M) \in \mathcal{O}(n^6)$. True
      iii) $timereq(M) \in \mathcal{O}(n^5/50)$. True
      iv) $timereq(M) \in \Theta(n^6)$. False
   b) Suppose that $timereq(M) = 5^n \cdot 3n^3$. Circle all of the following statements that are true:
      i) $timereq(M) \in \mathcal{O}(n^5)$. False
      ii) $timereq(M) \in \mathcal{O}(2^n)$. False
      iii) $timereq(M) \in \mathcal{O}(n!)$. True

2) Show a function f, from the natural numbers to the reals, that is $\mathcal{O}(1)$ but that is not constant.

$f(n) = 0$ if $n = 0$, else 1.

3) Assume the definitions of the variables given in the statement of Theorem 27.1. Prove that if $s > 1$ then:

$\mathcal{O}(n^t 2^n) \subseteq \mathcal{O}(2^{(n^s)})$.

Since $s > 1$, $s-1 > 0$. Since $t > 0$, $t+1 > 1$. So:

$$(t+1)^{\frac{1}{s-1}} > 1.$$

We'll let $k = \left\lceil (t+1)^{\frac{1}{s-1}} \right\rceil$ and $c = 1$. We'll show that $\forall n \geq k$, $1 \cdot (n^t 2^n) \leq 2^{(n^s)}$.

If $n \geq k$, then $n \geq 2$. So $\log_2 n \leq n$ and $\frac{\log_2 n}{n} \leq 1$. So:

[1] $$\frac{t \log_2 n}{n} + 1 \leq t + 1.$$

Also if $n \geq k$, then $n \geq (t+1)^{\frac{1}{s-1}}$ So:

[2] $n^{s-1} \geq t+1$.

Combining [1] and [2] we get:

[3] $$\frac{t \log_2 n}{n} + 1 \leq t + 1 \leq n^{s-1}.$$

From [3], multiplying by n, we get:

[4] $$t \log_2 n + n \leq n^s.$$

From [4], raising to the power of 2, we get:

[5] $2^{t\log_2 n} \cdot 2^n \le 2^{n^s}$.

But $2^{t\log_2 n} = n^t$. So, substituting into [5], we get:

[6] $(n^t 2^n) \le 2^{(n^s)}$.

4) Prove that, if $0 < a < b$, then $n^b \notin \mathcal{O}(n^a)$.

Suppose, to the contrary, that it were. Then there would exist constants k and c such that, for all $n \ge k$:

$$n^b \le c \cdot n^a.$$

Since $a < b$, $c^{\frac{1}{b-a}}$ exists and is positive. Choose:

$$n = \max(k, \left\lceil c^{\frac{1}{b-a}} \right\rceil + 1).$$

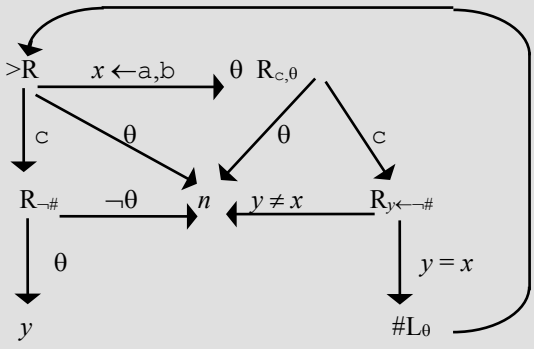
Then we have $n \ge k$ and $n > c^{\frac{1}{b-a}}$. So $n^{b-a} > c$ and $n^b > c \cdot n^a$. But this is a contradiction. So $n^b \notin \mathcal{O}(n^a)$.

5) Let M be the Turing machine shown in Example 17.9. M accepts the language WcW = $\{wcw : w \in \{a, b\}^*\}$. Analyze *timereq*($M$).

M is:

1. Loop:
   1.1. Move right to the first character. If it is c, exit the loop. Otherwise, overwrite it with θ and remember what it is.
   1.2. Move right to the c. Then continue right to the first unmarked character. If it is θ, halt and reject. (This will happen if the string to the right of c is shorter than the string to the left.) If it is anything else, check to see whether it matches the remembered character from the previous step. If it does not, halt and reject. If it does, mark it off with #.
   1.3. Move back leftward to the first θ.
2. There are no characters remaining before the c. Make one last sweep left to right checking that there are no unmarked characters after the c and before the first blank. If there are, halt and reject. Otherwise, halt and accept.

In our macro language, M is;



Let n be the length of M's input string w. We must determine the number of steps that M executes in the worst case. So we will not consider cases in which it executes the loop of statement 1 prematurely. So we will consider only strings of the form xcxy, where both x and $y \in \{a, b\}^*$. Note that xc$x \in W$cW.

In such cases, M will execute the complete loop of statement 1 $|x|$ times. On each iteration, it visits $|x| + 2$ squares as it moves to the right in statements 1.1 and 1.2. Then it visits $|x| + 1$ squares when it moves back to the left in statement 1.3. Then it enters the loop one last time and reads the c. The total number of steps required for this process is $|x|\cdot(2\cdot|x| + 3) + 1 = 2\cdot|x|^2 + 3\cdot|x| + 1$. After exiting the statement 1 loop, M must execute statement 2. To do that, it scans to the right $|x| + |y| + 1$ squares.

So the total time required by M is $(2\cdot|x|^2 + 3\cdot|x| + 1) + (|x| + |y| + 1) = 2\cdot|x|^2 + 4\cdot|x| + 2 + |y|$. So $timereq(M) = 2\cdot|x|^2 + 4\cdot|x| + 2 + |y|$.

We'd like to state $timereq(M)$ in terms of n. But we don't know how much of w is x and how much of it is y. We do know, though, that $n = 2\cdot|x| + |y| + 1$. So $timereq(M) = 2\cdot|x|^2 + 2\cdot|x| + 1 + n$. We also know that $x < n/2$.

So we can place an upper bound on the number of steps M executes as a function of n:

$$2\cdot|n/2|^2 + 2\cdot|n/2| + 1 + n = n^2/2 + n + 1.$$

6) Assume a computer that executes 10^{10} operations/second. Make the simplifying assumption that each operation of a program requires exactly one machine instruction. For each of the following programs P, defined by its time requirement, what is the largest size input on which P would be guaranteed to halt within a week?
   a) $timereq(P) = 5243n+649$.
   b) $timereq(P) = 5n^2$.
   c) $timereq(P) = 5^n$.

There are $60\cdot60\cdot24\cdot7 = 604{,}800 = 6.048\cdot10^5$ seconds in a week. So our computer executes $6.048\cdot10^{15}$ operations in a week.

a) We need to find the largest n such that $5243n+649 < 6.048\cdot10^{15}$. When $n = 1{,}153{,}538{,}050{,}734$, $5243n+649 = 6{,}047{,}999{,}999{,}999{,}011$. When $n = 1{,}153{,}538{,}050{,}735$, $5243n+649 = 6{,}048{,}000{,}000{,}004{,}254 > 6.048\cdot10^{15}$. So the largest size input on which P would be guaranteed to halt within a week is 1,153,538,050,734.

b) We need to find the largest n such that $5n^2 < 6.048\cdot10^{15}$. When $n = 34{,}779{,}304$, $5n^2 = 6{,}047{,}999{,}933{,}622{,}080$. When $n = 34{,}779{,}305$, $5243n+649 = 6{,}048{,}000{,}281{,}415{,}125$. So the largest size input on which P would be guaranteed to halt within a week is 34,779,304.

c) We need to find the largest n such that $5^n < 6.048\cdot10^{15}$. When $n = 22$, $5^n = 2{,}384{,}185{,}791{,}015{,}625$. When $n = 23$, $5^n = 11{,}920{,}928{,}955{,}078{,}125$. So the largest size input on which P would be guaranteed to halt within a week is 22.

7) Let each line of the following table correspond to a problem for which two algorithms, A and B, exist. The table entries correspond to $timereq$ for each of those algorithms. Determine, for each problem, the smallest value of n (the length of the input) such that algorithm B runs faster than algorithm A.

A	***B***
n^2	$572n + 4171$
n^2	$1000n \log_2 n$
$n!$	$450n^2$
$n!$	$3^n + 2$

- Row 1: When $n = 579$, $n^2 = 335{,}241$ and $572n + 4171 = 335{,}359$. When $n = 580$, $n^2 = 336{,}400$ and $572n + 4171 = 335{,}931$. So the smallest value of n such that algorithm B runs faster than algorithm A is 580.
- Row 2: When $n = 13{,}746$, $n^2 = 188{,}952{,}516$ and $1000n \log_2 n = 188{,}962{,}471.5$. When $n = 13{,}747$, $n^2 = 188{,}980{,}009$ and $1000n \log_2 n = 188{,}977{,}660.9$. So the smallest value of n such that algorithm B runs faster than algorithm A is 13,747.
- Row 3: When $n = 7$, $n! = 5{,}040$ and $450n^2 = 22{,}050$. When $n = 8$, $n! = 40{,}320$ and $450n^2 = 28{,}800$. So the smallest value of n such that algorithm B runs faster than algorithm A is 8.
- Row 4: When $n = 6$, $n! = 720$ and $3^n + 2 = 731$. When $n = 7$, $n! = 5{,}040$ and $3^n + 2 = 2{,}189$. So the smallest value of n such that algorithm B runs faster than algorithm A is 7.

8) Show that $L = \{<M>: M \text{ is a Turing machine and } timereq(M) \in \mathcal{O}(n^2)\}$ is not in SD.

The basic idea is that $timereq(M)$ is only defined if M halts on all inputs. We prove that L is not in SD by reduction from ¬H. Let R be a mapping reduction from ¬H to L defined as follows:

$R(<M, w>) =$
1. Construct the description $<M\#>$ of a new Turing machine $M\#(x)$ that, on input x, operates as follows:
   1.1. Run M on w for $|x|$ steps or until it halts naturally.
   1.2. If M would have halted naturally, then loop.
   1.3. Else halt.
2. Return $<M\#>$.

If *Oracle* exists and semidecides L, then $C = Oracle(R(<M, w>))$ semidecides ¬H:

- If $<M, w> \in$ ¬H: M does not halt on w, so $M\#$ will never discover that M would have halted. So, on all inputs, $M\#$ halts in $\mathcal{O}(n)$ steps (since the number of simulation steps it runs is n). So $timereq(M\#) \in \mathcal{O}(n^2)$ and $Oracle(<M\#>)$ accepts.
- If $<M, w> \notin$ ¬H: M halts on w. So, on some (long) inputs, $M\#$ will notice the halting. On those inputs, it fails to halt. So $timereq(M\#)$ is undefined and thus not in $\mathcal{O}(n^2)$. $Oracle(<M\#>)$ does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

9) Consider the problem of multiplying two $n \times n$ matrices. The straightforward algorithm *multiply* computes $C = A \cdot B$ by computing the value for each element of C using the formula:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} \text{ for } i, j = 1, \ldots, n.$$

Multiply uses n multiplications and n-1 additions to compute each of the n^2 elements of C. So it uses a total of n^3 multiplications and n^3-n^2 additions. Thus $timereq(multiply) \in \Theta(n^3)$.

We observe that any algorithm that performs at least one operation for each element of C must take at least n^2 steps. So we have an n^2 lower bound and an n^3 upper bound on the complexity of matrix multiplication. Because matrix multiplication plays an important role in many kinds of applications, the question naturally arose, "Can we narrow that gap?" In particular, does there exist a better than $\Theta(n^3)$ matrix multiplication algorithm? In [Strassen 1969], Volker Strassen showed that the answer to that question is yes.

Strassen's algorithm exploits a divide and conquer strategy in which it computes products and sums of smaller submatrices. Assume that $n = 2^k$, for some $k \geq 1$. (If it is not, then we can make it so by expanding the original matrix with rows and columns of zeros, or we can modify the algorithm presented here and divide the original matrix up differently.) We begin by dividing A, B, and C into 2×2 blocks. So we have:

$$A = \begin{bmatrix} A_{1,1} A_{1,2} \\ A_{2,1} A_{2,2} \end{bmatrix},\ B = \begin{bmatrix} B_{1,1} B_{1,2} \\ B_{2,1} B_{2,2} \end{bmatrix}, \text{ and } C = \begin{bmatrix} C_{1,1} C_{1,2} \\ C_{2,1} C_{2,2} \end{bmatrix},$$

where each $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$ is a $2^{k-1} \times 2^{k-1}$ matrix.

With this decomposition, we can state the following equations that define the values for each element of C:

$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

So far, decomposition hasn't bought us anything. We must still do eight multiplications and four additions, each of which must be done on matrices of size 2^{k-1}. Strassen's insight was to define the following seven equations:

$Q_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$
$Q_2 = (A_{2,1} + A_{2,2})B_{1,1}$
$Q_3 = A_{1,1}(B_{1,2} - B_{2,2})$
$Q_4 = A_{2,2}(B_{2,1} - B_{1,1})$
$Q_5 = (A_{1,1} + A_{1,2})B_{2,2}$
$Q_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$
$Q_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$

These equations can then be used to define the values for each element of C as follows:

$C_{1,1} = Q_1 + Q_4 - Q_5 + Q_7$
$C_{1,2} = Q_3 + Q_5$
$C_{2,1} = Q_2 + Q_4$
$C_{2,2} = Q_1 - Q_2 + Q_3 + Q_6$

Now, instead of eight matrix multiplications and four matrix additions, we do only seven matrix multiplications, but we must also do eighteen matrix additions (where a subtraction counts as an addition). We've replaced twelve matrix operations with 25. But matrix addition can be done in $\mathcal{O}(n^2)$ time, while matrix multiplication remains more expensive.

Strassen's algorithm applies these formulas recursively, each time dividing each matrix of size 2^k into four matrices of size 2^{k-1}. The process halts when $k = 1$. (Efficient implementations of the algorithm actually stop the recursion sooner and use the simpler *multiply* procedure on small submatrices. We'll see why in part (e) of this problem.) We can summarize the algorithm as follows:

Strassen(A, B, k: where A and B are matrices of size 2^k) =
    If $k = 1$ then compute the Q's using scalar arithmetic. Else, compute them as follows:
    Q_1 = *Strassen*($(A_{1,1} + A_{2,2})$, $(B_{1,1} + B_{2,2})$, k-1).
    Q_2 = *Strassen*($(A_{2,1} + A_{2,2})$, $B_{1,1}$, k-1).
        … /* Compute all the Q matrices as described above.
    Q_7 =
    $C_{1,1}$ =
        … /* Compute all the C matrices as described above.
    $C_{2,2}$ =
    Return C.

In the years following Strassen's publication of his algorithm, newer ones that use even fewer operations have been discovered 💻. The fastest known technique is the Coppersmith-Winograd algorithm, whose time complexity is $\mathcal{O}(n^{2.376})$. But it too complex to be practically useful. There do exist algorithms with better performance than *Strassen*, but it opened up this entire line of inquiry, so we should understand its complexity. In this problem, we will analyze *timereq* of *Strassen* and compare it to *timereq* of the standard algorithm *multiply*. We shold issue one caveat before we start, however: The analysis that we are about to do just counts scalar multiplies and adds. It does not worry about such things as the behavior of caches and the use of pipelining. In practice, it turns out that the crossover point for *Strassen* relative to *multiply* 💻 is lower than our results suggest.

a) We begin by defining $mult'(k)$ to be the number of scalar multiplications that will be performed by *Strassen* when it multiplies two $2^k \times 2^k$ matrices. Similarly, let $add'(k)$ be the number of scalar additions. Describe both $mult'(k)$ and $add'(k)$ inductively by stating their value for the base case (when $k = 1$) and then describing their value for $k > 1$ as a function of their value for k-1.

$mult'(k) =$ if $k = 1$ then 7.
if $k > 1$ then $7 \cdot mult'(k\text{-}1)$

To describe $add'(k)$, we consider the additions that are done when when *Strassen* is called the first time, plus the additions that are done when it is called recursively to do the multiplies. Adding two matrices of size n take n^2 additions. So we have:

$add'(k) =$ if $k = 1$ then 18.
if $k > 1$ then $18 \cdot (2^{k-1})^2 + 7 \cdot add'(k\text{-}1)$

b) To find closed form expressions for $mult'(k)$ and add' requires solving the recurrence relations that were given as answers in part (a). Solving the one for $mult'(k)$ is easy. Solving the one for $add'(k)$ is harder. Prove that the following are correct:

$mult'(k) = 7^k$.
$add'(k) = 6 \cdot (7^k - 4^k)$.

We first show how we derived the claim about $add'(k)$:

For $k > 1$, we have:

$$\begin{aligned}
add'(k) &= 18 \cdot (2^{k-1})^2 + 7 \cdot add'(k\text{-}1) \\
&= 18 \cdot 4^{k-1} + 7 \cdot add'(k\text{-}1) && \text{Writing out one more step:} \\
&= 18 \cdot 4^{k-1} + 7 \cdot (18 \cdot 4^{k-2} + 7 \cdot add'(k\text{-}2)) \\
&= 18 \cdot 4^{k-1} + 7 \cdot 18 \cdot 4^{k-2} + 7^2 \cdot add'(k\text{-}2) && \text{Writing out additional steps:} \\
&\quad \dots \\
&= 18 \cdot 4^{k-1} + 7 \cdot 18 \cdot 4^{k-2} + 7^2 \cdot 18 \cdot 4^{k-3} + \dots + 7^{k-1} \cdot add'(1) \\
&= 18 \cdot 4^{k-1} + 7 \cdot 18 \cdot 4^{k-2} + 7^2 \cdot 18 \cdot 4^{k-3} + \dots + 7^{k-1} \cdot 18 \\
&= 18 \cdot (4^{k-1} + 7 \cdot 4^{k-2} + 7^2 \cdot 4^{k-3} + \dots + 7^{k-1}) \\
&= 18 \cdot 4^{k-1} \cdot \left(1 + \frac{7}{4} + \left(\frac{7}{4}\right)^2 + \dots + \left(\frac{7}{4}\right)^{k-1}\right) && \text{Using (*) defined below:} \\
&= 18 \cdot 4^{k-1} \cdot \left(\frac{\left(\frac{7}{4}\right)^k - 1}{\frac{7}{4} - 1}\right)
\end{aligned}$$

$$= 18 \cdot 4^{k-1} \cdot \left(\frac{\frac{7^k}{4^k} - 1}{\frac{3}{4}} \right)$$

$$= 24 \cdot \left(\frac{7^k}{4} - \frac{4^k}{4} \right)$$

$$= 6 \cdot (7^k - 4^k)$$

Where (*) is the fact that: $\sum_{i=0}^{k-1} a^i = \frac{a^k - 1}{a - 1}$

We can prove the claim by induction on k. The base case is $k = 1$. $Add'(k) = 18$. Assume that $add'(k) = 6 \cdot (7^k - 4^k)$. We show that the claim must then be true for $k+1$:

$$\begin{aligned} add'(k+1) &= 18 \cdot (2^k)^2 + 7 \cdot add'(k) \\ &= 18 \cdot 4^k + 7 \cdot 6 \cdot (7^k - 4^k) \\ &= 6 \cdot (3 \cdot 4^k + 7 \cdot (7^k - 4^k)) \\ &= 6 \cdot (3 \cdot 4^k + 7^{k+1} - 7 \cdot 4^k) \\ &= 6 \cdot (7^{k+1} - 7 \cdot 4^k + 3 \cdot 4^k) \\ &= 6 \cdot (7^{k+1} - 4 \cdot 4^k) \\ &= 6 \cdot (7^{k+1} - 4^{k+1}) \end{aligned}$$

c) We'd like to define the time requirement of *Strassen*, when multiplying two $n \times n$ matrices, as a function of n, rather than as a function of $\log_2 n$, as we have been doing. So define $mult(n)$ to be the number of multiplications that will be performed by *Strassen* when it multiplies two $n \times n$ matrices. Similarly, let $add(n)$ be the number of additions. Using the fact that $k = \log_2 n$, state $mult(n)$ and $add(n)$ as functions of n.

$mult(n) = 7^{\log_2 n}$.
$add(n) = 6 \cdot (7^{\log_2 n} - 4^{\log_2 n})$.

d) Determine values of α and β, each less than 3, such that $mult(k) \in \Theta(n^\alpha)$ and $add(k) \in \Theta(n^\beta)$.

We use the fact that, for any a and b, $a^{\log b} = b^{\log a}$. So:
$mult(n) = 7^{\log_2 n} = n^{\log_2 7}$. Thus $\alpha = \log_2 7 = 2.83$.
$add(n) = 6 \cdot (7^{\log_2 n} - 4^{\log_2 n}) = 6 \cdot (n^{\log_2 7} - n^{\log_2 4}) = 6 \cdot (n^{\log_2 7} - n^2)$. Thus $\beta = 2.83$.

e) Let $ops(n) = mult(n) + add(n)$ be the total number of scalar multiplications and additions that *Strassen* performs to multiply two $n \times n$ matrices. Recall that, for the standard algorithm *multiply*, this total operation count is $2n^3 - n^2$. We'd like to find the crossover point, i.e., the point at which *Strassen* performs fewer scalar operations than *multiply* does. So find the smallest value of k such that $n = 2^k$ and $ops(n) < 2n^3 - n^2$. (Hint: Once you have an equation that describes the relationship between the operation counts of the two algorithms, just start trying candidates for k, starting at 1.)

$ops'(k) = mult'(k) + add'(k)$
$= 7^k + 6 \cdot (7^k - 4^k)$

Stated in terms of k, the number of operations executed by *multiply* is:

$2 \cdot (2^k)^3 - (2^k)^2 = 2 \cdot 8^k - 4^k$

So we need to find the smallest k such that:

$$7^k + 6\cdot(7^k - 4^k) \le 2\cdot 8^k - 4^k$$
$$7^k + 6\cdot(7^k - 4^k) - 2\cdot 8^k + 4^k \le 0$$
$$7\cdot 7^k - 5\cdot 4^k - 2\cdot 8^k \le 0$$

Trying candidates for k, starting at 1, we find the smallest value that satisfies the inequality is 10.

10) In this problem, we will explore the operation of the Knuth-Morris-Pratt string search algorithm that we described in Example 27.5. Let p be the pattern `cbacbcc`.

a) Trace the execution of *buildoverlap* and show the table T that it builds.

We show T, as well as the kernels that correspond to each of its elements:

j	0	1	2	3	4	5	6
$T[j]$	-1	0	0	0	1	2	1
the kernel	ε	ε	`b`	`ba`	`bac`	`bcab`	`bacbc`

b) Using T, trace the execution of *Knuth-Morris-Pratt*(`cbaccbacbcc`, `cbacbcc`).

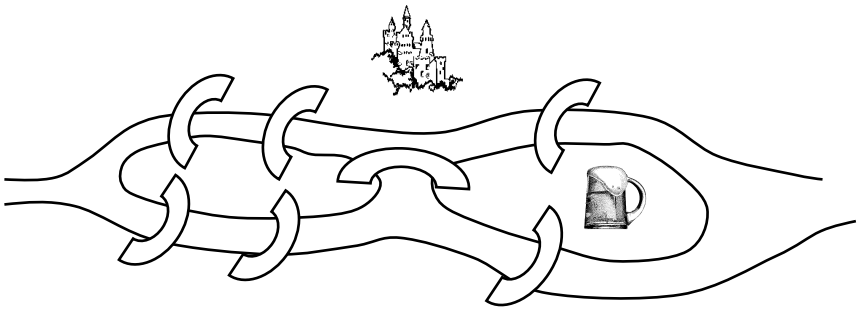
```
cbaccbacbcc
cbacbcc
    ×

cbaccbacbcc
   cbacbcc
    ×

cbaccbacbcc
    cbacbcc
```

28 Time Complexity Classes

1) In Section 28.1.5, we described the Seven Bridges of Königsberg. Consider the following modification:



The good prince lives in the castle. He wants to be able to return home from the pub (on one of the islands as shown above) and cross every bridge exactly once along the way. But he wants to make sure that his evil twin, who lives on the other river bank, is unable to cross every bridge exactly once on his way home from the pub. The good prince is willing to invest in building one new bridge in order to make his goal achievable. Where should he build his bridge?

There will be an Eulerian path from the pub to the castle if the vertex corresponding to the location of the pub and the vertex corresponding to the location of the castle each have odd degree and all other vertices have even degree. To make this true, add a bridge from the castle side of the river to the non-pub island. The vertex corresponding to the non-castle side of the river now has even degree, so there is no Eulerian circuit that ends there. Thus the evil twin cannot get home from the pub by traversing all the bridges.

2) Consider the language NONEULERIAN = {<*G*> : *G* is an undirected graph and *G* does not contain an Eulerian circuit}.
   a) Show an example of a connected graph with 8 vertices that is in NONEULERIAN.

   Anything that has any vertices of odd degree.

   b) Prove that NONEULERIAN is in P.

   NONEULERIAN is the complement of EULERIAN, which we showed is in P. The class P is closed under complement.

3) Show that each of the following languages is in P:
   a) WWW = {*www*: *w* ∈ {a, b}*}.

   The following deterministic polynomial-time algorithm decides WWW. On input *s* do:

   1. Make one pass through *s*, counting the number of symbols it contains. If |*s*| is not divisible by 3, reject.
   2. Make another pass through *s*, leaving the first |*s*|/3 symbols on tape 1, copying the second |*s*|/3 symbols onto tape 2, and the third |*s*|/3 symbols onto tape 3.
   3. Make one sweep through the three tapes making sure that they are the same.
   4. If they are, accept. Else reject.

b) {<*M*, *w*> : Turing machine *M* halts on *w* within 3 steps}.

The following deterministic polynomial-time algorithm decides {<*M*, *w*> : Turing machine *M* halts on *w* within 3 steps}. On input <*M*, *w*> do:

1. Simulate M on w for three steps.
2. If the simulation halted, accept. Else reject.

In Section 17.7 we describe the operation of a Universal Turing machine *U* that can do the simulation that we require. If *k* is the number of simulated steps, *U* takes $\mathcal{O}(|M| \cdot k)$ steps, which is polynomial in |<*M*, *w*>|.

c) EDGE-COVER = {<*G, k*>: *G* is an undirected graph and there exists an edge cover of *G* that contains at most *k* edges}.

4) In the proof of Theorem B.2, we present the algorithm *3-conjunctiveBoolean*, which, given a Boolean wff *w* in conjunctive normal form, constructs a new wff *w′*, where *w′* is in 3-CNF.

a) We claimed that *w′* is satisfiable iff *w* is. Prove that claim.

Recall:

3-conjunctiveBoolean(*w*: wff of Boolean logic) =

1. If, in *w*, there are any clauses with more than three literals, split them apart, add additional variables as necessary, and form a conjunction of the resulting clauses. Specifically, if $n > 3$ and there is a clause of the following form:

   $(l_1 \vee l_2 \vee l_3 \vee \ldots \vee l_n)$,

   then it will be replaced by the following conjunction of n-2 clauses that can be constructed by introducing a set of literals Z_1 - Z_{n-3} that do not otherwise occur in the formula:

   $(l_1 \vee l_2 \vee Z_1) \wedge (\neg Z_1 \vee l_3 \vee Z_2) \wedge \ldots \wedge (\neg Z_{n-3} \vee l_{n-1} \vee l_n)$

2. If there is any clause with only one or two literals, replicate one of those literals once or twice so that there is a total of three literals in the clause.

We show that both steps preserve satisfiablity. Note that *w* is satisfiable iff every clause *C* in *w* is satisfiable. So we can consider separately the satisfiabilty of each clause.

Step 1: Let *C* be any clause in *w* that is changed by step 1. Call *C*'s replacement *C′*. Note that, if *C* contains *n* literals, then *C′* contains *n*-2 clauses.

- Suppose that *C* is satisfiable. Then there is some assignment *A* of truth values to the variables of *C* that makes *C True*. *A* must make at least one literal of *C True*. Pick one such *True* literal. Call it l_k. Then there exists an assignment *A′* of truth values to the variables of *C′* such that *C′* is also *True*. We construct *A′* as follows: Assign to all the variables in *C* the values that *A* assigns to them. Make Z_1 – Z_{k-2} *True* . This makes clauses 1 through *k*-2 of *C′True*. Make Z_{k-1} – Z_{n-3} *False*. This makes clauses *k*-1 through *n*-2 of *C′True*. Clause *k* of *C′* is *True* because l_k is *True*. So *C′* is satisfiable if *C* is.
- Suppose that *C* is not satisfiable. Then every assignment *A* makes *C False*. In this case, we can show that every assignment *A′* of truth values to the variables of *C′* also makes *C′False*. *A′* must be the extension of some assignment *A* to the variables of *C*. (In other words, it assigns to all the variables of *C* the same values *A* does. Then it also assigns values to the new variables of *C′*.) We know that every *A* makes *C False*. Thus it must make every literal in *C False*. Now consider *C′*. Any assignment *A′* that makes *C′True* must make each of its clauses *True*. The only way to do that is with the *Z*'s, since all the original literals are made *False* by the assignment *A* of which *A′* is an extension. Each *Z* can only make one clause *True*. There are *n*-2 clauses that need to be made *True*. But there are only *n*-3

Z's. So there is no way to make all the clauses of C' *True* with a single assignment A'. So C' is unsatisfiable if C is.

Step 2 preserves equivalence and thus satisfiability.

b) Prove that *3-conjunctiveBoolean* runs in polynomial time.

3-conjunctiveBoolean examines each clause of w. So the process described above is done $\mathcal{O}(|w|)$ times. Step 2 of the process takes constant time.

Step 1: The time required to build C'' from C is linear in the length of C''. If C contains n literals, then C' contains n-2 clauses. Each clause has constant length. So the length of C'' is $\mathcal{O}(n)$, which is $\mathcal{O}(|w|)$.

So the total time is $\mathcal{O}(|w|^2)$.

5) Consider the language 2-SAT = {<w> : w is a wff in Boolean logic, w is in 2-conjunctive normal form and w is satisfiable}.

a) Prove that 2-SAT is in P. (Hint: use resolution, as described in B.1.2.)

Let w be a wff in 2-conjunctive normal form. Let v be the number of Boolean variables in w. There are $v^4/2$ possible different 2-CNF clauses using v variables (since each variable in the clause may be any of the v variables, either negated or not).

If w is unsatisfiable, then, if we apply resolution to the clauses of w, we'll derive *nil*. Whenever two 2-CNF clauses are resolved, the resolvent is also a 2-CNF clause. So begin resolving the clauses of w with each other and then with any new resolvent clauses, making sure not to add to the list any resolvent clause that is already there. This process can go on for at most $v^4/2$ steps, since that's the number of distinct possible clauses. If *nil* is derived at any point, halt and report that w is not satisfiable. If all possible resolution steps have been done and *nil* has not been derived, then halt and report that w is satisfiable. This polynomial-time algorithm proves that 2-SAT is in P.

b) Why cannot your proof from part a) be extended to show that 3-SAT is in P?

The key to that proof was the observation that, when 2 2-CNF clauses are resolved, another 2-CNF clause is created. It is not true that, when 2 3-CNF clauses are resolved, another 3-CNF clause is created. Longer ones will arise. So we can no longer place a polynomial bound on the number of clauses we'd have to try before we could guarantee that *nil* cannot be produced.

c) Now consider a modification of 2-SAT that might, at first, seem even easier, since it may not require all of the clauses of w to be simultaneously satisfied. Let 2-SAT-MAX = {<w, k> : w is a wff in Boolean logic, w is in 2-conjunctive normal form, $1 \le k \le |C|$, where $|C|$ is the number of clauses in w, and there exists an assignment of values to the variables of w that simultaneously satisfies at least k of the clauses in w}. Show that 2-SAT-MAX is NP-complete.

By reduction from 3-SAT. For the details, see [Garey and Johnson 1979].

6) In Chapter 9, we showed that all of the questions that we posed about regular languages are decidable. We'll see, in Section 29.3.3, that while decidable, some straightforward questions about the regular languages appear to be hard. Some are easy however. Show that each of the following languages is in P:

a) DFSM-ACCEPT = {<*M*, *w*> : *M* is a DFSM and $w \in L(M)$}.

DFSM-ACCEPT can be decided by the algorithm *dfsmsimulate* that we presented in Section 5.7.1. Recall:

dfsmsimulate(*M*: DFSM, *w*: string) =
1. *st* = *s*.
2. Repeat:
   2.1. *c* = get-next-symbol(*w*).
   2.2. If *c* ≠ end-of-file then:
        st = δ(*st*, *c*).
   until *c* = end-of-file.
3. If *st* ∈ *A* then accept else reject.

It is straightforward to implement the δ lookup step in $\mathcal{O}(|M|)$ time. The outer loop must be executed $|w|+1$ times. So $timereq(dfsmsimulate) \in \mathcal{O}(|<M, w>|^2)$.

b) FSM-EMPTY = {<*M*> : *M* is a FSM and $L(M) = \varnothing$}.

In Section 9.1.2, we presented three algorithms for deciding, given an FSM *M*, whether $L(M) = \varnothing$. One of them, *emptyFSMgraph*, runs in polynomial time. Recall:

emptyFSMgraph(*M*: FSM) =
1. Mark all states that are reachable via some path from the start state of *M*.
2. If at least one marked state is an accepting state, return *False*. Else return *True*.

Step 1 can be implemented using the same polynomial-time marking algorithm that we used to show that CONNECTED is in P.

c) DFSM-ALL = {<*M*> : *M* is a DFSM and $L(M) = \Sigma^*$}.

In Section 9.1.2, we presented the algorithm *totalFSM*, which decides DFSM-ALL. Recall:

totalFSM(*M*: FSM) =
1. Construct *M'* to accept $\neg L(M)$.
2. Return *emptyFSM*(*M'*).

If *M* is deterministic, then step 1 can be implemented in polynomial time. (It must simply search the description of *M*, and set *A* to *K*-*A*). Step 2 can be implemented in polynomial time, as described in the solution to part (b).

7) We proved (in Theorem 28.1) that P is closed under complement. Prove that it is also closed under:

a) Union.

If L_1 and L_2 are in P, then there exist deterministic, polynomial-time Turing machines M_1 and M_2 that decide them. We show a new, deterministic, polynomial-time Turing machine *M* that decides $L_1 \cup L_2$. On input *w*, run M_1 on *w*. If it accepts, accept. Otherwise, run M_2 on *w*. If it accepts, accept. Otherwise reject.

b) Concatenation.

If L_1 and L_2 are in P, then there exist deterministic, polynomial-time Turing machines M_1 and M_2 that decide them. We show a new, deterministic, polynomial-time Turing machine M that decides $L_1 L_2$. There are $|w|+1$ ways to divide a string w into two substrings. So M works as follows on input w:

1. For $i := 1$ to $|w|+1$ do:
   1.1. Divide w into two substrings at position i. Call the first s_1 and the second s_2.
   1.2. Run M_1 on s_1.
   1.3. If it accepts, run M_2 on s_2. If it accepts, accept.
2. Reject (since none of the ways of dividing w into two strings that meet the requirement was found).

The time required to run M is $(|w|+1)\cdot(timereq(M_1) + timereq(M_2))$, which is polynomial since $timereq(M_1)$ and $timereq(M_2)$ are.

c) Kleene star.

8) It is not known whether NP is closed under complement. But prove that it is closed under:
a) Union.

If L_1 and L_2 are in NP, then there exist nondeterministic, polynomial-time Turing machines M_1 and M_2 that decide them. We show a new, nondeterministic, polynomial-time Turing machine M that decides $L_1 \cup L_2$. On input w, run M_1 on w. If it accepts, accept. Otherwise, run M_2 on w. If it accepts, accept. Otherwise reject.

b) Concatenation.

If L_1 and L_2 are in NP, then there exist nondeterministic, polynomial-time Turing machines M_1 and M_2 that decide them. We show a new, nondeterministic, polynomial-time Turing machine M that decides L_1L_2. On input w, nondeterministically divide w into two pieces. Run M_1 on the first piece and M_2 on the second. If they both accept, accept.

c) Kleene star.

Similarly to (b), except nondeterministically choose how many pieces to cut w into.

9) If L_1 and L_2 are in P and $L_1 \subseteq L \subseteq L_2$, must L be in P? Prove your answer.

No. Let $L_1 = \varnothing$. Let L_2 be the set of all syntactically correct logical formulas in the language of Presburger arithmetic. Let L be the set of theorems of Presburger arithmetic. We know that any algorithm to decide L requires exponential time.

10) Show that each of the following languages is NP-complete by first showing that it is in NP and then showing that it is NP-hard:
a) CLIQUE = {<G, k> : G is an undirected graph with vertices V and edges E, k is an integer, $1 \le k \le |V|$, and G contains a k-clique}.

Given <G, k> and a proposed certificate c, a deterministic polynomial-time verifier first checks that c contains k vertices. If it does not, it rejects. If it does, it checks that each pair of vertices in c is connected by an edge in G. If there's a pair that is not, it rejects. It takes $\mathcal{O}(|c|^2)$ time to do that check.

We prove that CLIQUE is NP-hard by reduction from 3-SAT. Let E be a Boolean expression in 3-conjunctive normal form. E is the conjunction of clauses, each of which has the form $l_1 \vee l_2 \vee l_3$. We define a reduction R from SAT to CLIQUE that, on input <E>, builds a graph G as follows:

1. For each clause c in E build a set of seven vertices, one for each of the satisfying assignments of values to the variables of c. (We can make just any one of the individuals true, make any pair of two of them true, or make all three of them true.) Make no edges between any pair of these vertices.
2. Consider all of the assignment nodes created in step 1. Add to G an edge between every pair with the property that both elements come from different clauses and the assignments to which they correspond do not conflict.

Let k be the number of clauses in E. No pair of nodes from the same clause can occur in any clique. Since there are k such independent clause groups, the only way for there to be a clique in G of size k is for it to contain one node from each clause group. It can only do that if there is a way to pick nodes, one from each clause group, that correspond to assignments that don't conflict. So there is a k-clique in G iff there is a satisfying assignment for E.

b) SUBSET-SUM = {<S, k> : S is a multiset (i.e., duplicates are allowed) of integers, k is an integer, and there exists some subset of S whose elements sum to k}.

A deterministic, polynomial-time verifier V checks the length of a proposed certificate c. If it is longer than $|S|$, it rejects. Otherwise it sums the elements of c. If the sum is equal to k it accepts, othewise it rejects. V runs in polynomial time because it can check the length of c in $\mathcal{O}(|S|)$ time. It can sum the elements of c in $\mathcal{O}(|c|)$ time (assuming constant cost per addition). And it can compare that sum to k in $\mathcal{O}(k)$ time. So timereq(V) $\in \mathcal{O}(|<S, k>|)$.

It remains to show that SUBSET-SUM is NP hard.

c) SET-PARTITION = {<S> : S is a multiset (i.e., duplicates are allowed) of objects, each of which has an associated cost, and there exists a way to divide S into two subsets, A and $S - A$, such that the sum of the costs of the elements in A equals the sum of the costs of the elements in $S - A$}.

A deterministic, polynomial-time verifier V checks the length of a proposed certificate c (which will be interpreted as a proposal for the subset A). If $|c| > |S|$, V rejects. Otherwise it behaves as follows:

1. Go through c and mark off the corresponding element of S.
2. Go through S and compute two sums: the cost of all of the marked elements and the cost of all of the unmarked elements.
3. Compare the two sums. If they are equal, accept. Otherwise reject.

V runs in polynomial time because it can check the length of c in $\mathcal{O}(|S|)$ time. If it continues, then $\mathcal{O}(|c|) \subseteq \mathcal{O}(|S|)$. It can execute step 1 in $\mathcal{O}(|c|)$ time. It can execute step 2 in $\mathcal{O}(|S|)$ time. And it can execute step 3 in $\mathcal{O}(S)$ time. So timereq(V) $\in \mathcal{O}(|<S>|)$.

It remains to show that SUBSET-SUM is NP hard.

d) KNAPSACK = {<S, v, c> : S is a set of objects each of which has an associated cost and an associated value, v and c are integers, and there exists some way of choosing elements of S (duplicates allowed) such that the total cost of the chosen objects is at most c and their total value is at least v}.

e) LONGEST-PATH = {<*G, u, v, k*>: *G* is an unweighted, undirected graph, *u,* and *v* are vertices in *G, k* ≥ 0, and there exists a path with no repeated edges from *u* to *v* whose length is at least *k*}.

A nondeterministic, polynomial-time decider *M* for LONGEST-PATH works as follows:

1. Initialize *path* to *u*.
2. Until the last vertex in *path* is *v* or there are no vertices left to choose do:
   2.1. Choose an unmarked vertex *x* from the vertices of *G*.
   2.2. Mark *x*.
   2.3. Add *x* to the end of *path*.
3. If the length of *path* is at least *k*, accept. Otherwise reject.

Steps 2.1 - 2.3 can be executed in $\mathcal{O}(|<G>|)$ time. The maximum number of times through the step 2 loop is the number of vertices in *G*. Steps 1 and 3 take constant time. So $timereq(M) \in \mathcal{O}(|<G>|^2)$.

It remains to show that SUBSET-SUM is NP hard.

f) BOUNDED-PCP = {<*P*, *k*> : *P* is an instance of the Post Correspondence problem that has a solution of length less than or equal to *k*}.

Nondeterministically choose a string of indices of length less than or equal to *k*. Check to see if it works.

It remains to show that BOUNDED-PCP is NP-hard.

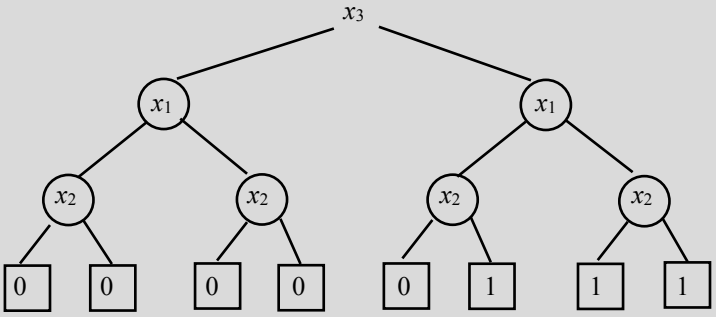
11) Let USAT = {<*w*> : *w* is a wff in Boolean logic and *w* has exactly one satisfying assignment}. Does the following nondeterministic, polynomial-time algorithm decide USAT? Explain your answer.

decideUSAT(<*w*>) =
1. Nondeterministically select an assignment *x* of values to the variables in *w*.
2. If *x* does not satisfy *w*, reject.
3. Else nondeterministically select another assignment $y \neq x$.
4. If *y* satisfies *w*, reject.
5. Else accept.

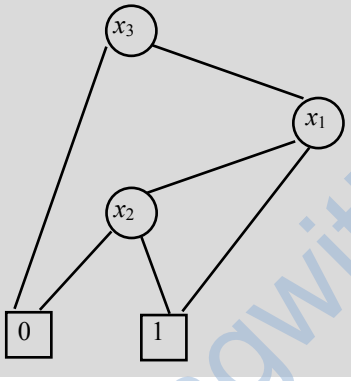
No. Suppose that *w* has exactly two satisfying assignments, *a* and *b*. In this case, *decideUSAT* should reject. But, on one of its branches, it will try *a* and *c*, where $c \neq b$. Since *a* is a satisfying assignment and *c* isn't, it will accept.

12) Ordered binary decision diagrams (OBDDs) are useful in manipulating Boolean formulas such as the ones in the language SAT. They are described in b.1.3. Consider the Boolean function f_1 shown there. Using the variable ordering ($x_3 < x_1 < x_2$), build a decision tree for *f*. Show the (reduced) OBDD that *createOBDDfromtree* will create for that tree.

The original decision tree is:



The (reduced) OBDD is:



13) Complete the proof of Theorem 28.18 by showing how to modify the proof of Theorem 28.16 so that *R* constructs a formula in conjunctive normal form. Show that *R* still runs in polynomial time.

See [Sipser 2006].

14) Show that, if P = NP, then there exists a deterministic, polynomial-time algorithm that finds a satisfying assignment for a Boolean formula if one exists.

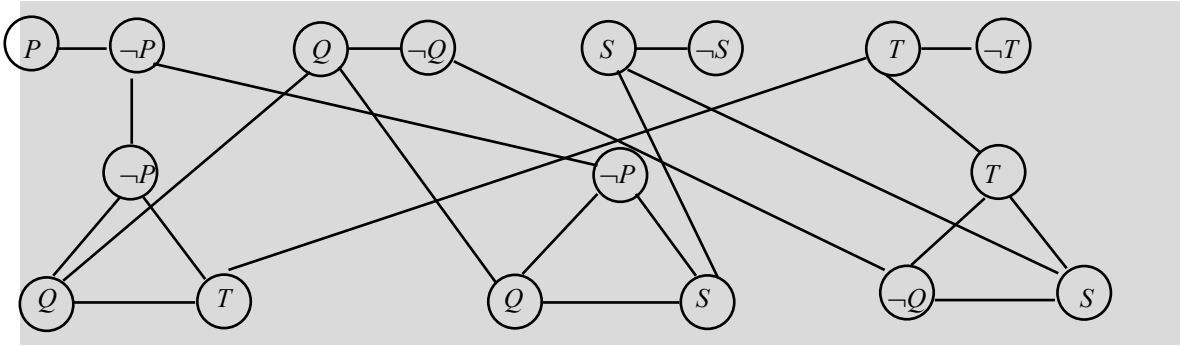
If P = NP, then, since SAT is in NP, it is also in P Thus there exists a deterministic, polynomial-time algorithm *SATverf*(<*w*>) that decides whether *w* is satisfiable. Using it, we define the following polynomial-time algorithm that finds a satisfying assignment if one exists:

SATsolve(<*w*>) =

1. If *v* contains no variables, then return *nil*, an empty list of substitutions.
2. Otherwise, choose one variable *v* in *w*.
3. Replace every instance of *v* in *w* by *True*. Call the resulting formula *w′*.
4. If *SATverf*(<*w′*>) accepts, then return the result of appending *True*/*v* to the result that is returned by *SATsolve*(<*w′*>).
5. Otherwise, replace every instance of *v* in *w* by *False*. Call the resulting formula *w′*.
6. If *SATverf*(<*w′*>) accepts, then return the result of appending *False*/*v* to the result that is returned by *SATsolve*(<*w′*>).
7. Otherwise, no satisfying assignment exists. Return *Error*.

SATsolve will execute one recursive step for each variable in *w*. So if *SATverf* runs in polynomial time, so does *SATsolve*.

15) Let R be the reduction from 3-SAT to VERTEX-COVER that we defined in the proof of Theorem 28.20. Show the graph that R builds when given the Boolean formula, $(\neg P \vee Q \vee T) \wedge (\neg P \vee Q \vee S) \wedge (T \vee \neg Q \vee S)$.



16) We'll say that an assignment of truth values to variables ***almost satisfies*** a CNF Boolean wff with k clauses iff it satisfies at least k-1 clauses. A CNF Boolean wff is ***almost satisfiable*** iff some assignment almost satisfies it. Show that the following language is NP-complete:

- ALMOST-SAT = {<w> : w is an almost satisfiable CNF Boolean formula}.

ALMOST-SAT is in NP because an almost satisfying assignment is a certificate for it.

We'll show that ALMOST-SAT is NP-hard by reduction from SAT. Define the reduction $R(w) = w \wedge x \wedge \neg x$, where x is a variable that is not in w. Let m be the number of clauses in w. Then $R(w)$ has m+2 clauses. If w is satisfiable, then $R(w)$ has an almost satisfying assignment that is the satisfying assignment for w plus any assignment of value to x. If w is not satisfiable, then at most m-1 of its clauses can be satisfied by any assignment. At most one of x and $\neg x$ can be satisfied by any given assignment. So at most m clauses of $R(w)$ can be satisfied. Thus it has no almost satisfying assignment.

17) Show that VERTEX-COVER is NP-complete by reduction from INDEPENDENT-SET.

See [Hopcroft, Motwani and Ullman 2001].

18) In Appendix O, we describe the regular expression sublanguage in Perl. Show that regular expression matching in Perl (with variables allowed) is NP-hard.

19) In most route-planning problems the goal is to find the shortest route that meets some set of conditions. But consider the following problem (aptly named the taxicab ripoff problem in [Lewis and Papadimitriou 1998]): Given a directed graph G with positive costs attached to each edge, find the longest path from vertex i to vertex j that visits no vertex more than once.
   a) Convert this optimization problem to a language recognition problem.
   b) Make the strongest statement you can about the complexity of the resulting language.

20) In Section 22.3, we introduced a family of tiling problems and defined the language TILES. In that discussion, we considered the question, "Given an infinite set T of tile designs and an infinite number of copies of each such design, is it possible to tile every finite surface in the plane?" As we saw there, that unbounded version of the problem is undecidable. Now suppose that we are again given a set T of tile designs. But, this time, we are also given n^2 specific tiles drawn from that set. The question we now wish to answer is, "Given a particular stack of n^2 tiles, is it possible to tile an $n \times n$ surface in the plane?" As before, the rules are that tiles may not be rotated or flipped and the abutting regions of every pair of adjacent tiles must be the same color. So, for example, suppose that the tile set is:

Then a 2 × 2 grid can be tiled as:

a) Formulate this problem as a language, FINITE-TILES.

FINITE-TILES = {<T, S> : ∃n (S is a set of n^2 tiles drawn from the tile set T and S can be used to tile an $n \times n$ surface in the plane}.

b) Show that FINITE-TILES is in NP.

Consider the following procedure *check*:

Assume that c, a proposed certificate for a string of the form <T, S>, is a list of tiles. If the length of c is n^2, for some integer n, then the first n elements of c will be interpreted as describing the tiles in row 1. The next n elements will describe the tiles in row 2, and so forth. The last n tiles will describe the tiles in row n.

On input <T, S, c> do:

1. Check that the length of c is equal to n^2 for some integer n. If it is not, reject.
2. Check that every tile in S is drawn from the tile set T. If not, reject.
3. Check that c uses each tile in S exactly once. If not, reject.
4. Check that c places tiles according to the rules for the language TILES. If it does not, reject.
5. If c has passed all of these tests, accept.

Check is a polynomial-time verifier for FINITE-TILES because:

- Step 1 can be done in polynomial time by computing the square root of $|c|$.
- Step 2 can be done by going through each tile in S and comparing it the list T. The time required to do this is $\mathcal{O}(|S| \cdot |T|)$.
- Step 3 can be done by going through the elements of c one at a time. For each, go through the elements of S until a match is found and mark it off. The time required to do this is $\mathcal{O}(|c| \cdot |S|)$.
- Step 4 requires looking at each boundary between a pair of tiles. The number of such boundaries is less than $4 \cdot |c|$ and it takes constant time to check each boundary. So the time required to do this is $\mathcal{O}(|c|)$.

c) Show that FINITE-TILES is NP-complete (by showing that it is NP-hard).

21) In Section 28.7.6, we defined what we mean by a map coloring.
   a) Prove the claim, made there, that a map is two-colorable iff it does not contain any point that is the junction of an odd number of regions. (Hint: use the pigeonhole principle.)
   b) Prove that 3-COLORABLE = {<m> : m is a 3-colorable map} is in NP.
   c) Prove that 3-COLORABLE = {<m> : m is a 3-colorable map} is NP-complete.

22) Define the following language:

- BIN-OVERSTUFFED = {<S, c, k> : S is a set of objects each of which has an associated size and it is *not* possible to divide the objects so that they fit into k bins, each of which has size c}.

Explain why it is generally believed that BIN-OVERSTUFFED is not NP-complete.

BIN-OVERSTUFFED is the complement of BIN-PACKING, which is NP-complete. No language has ever been discovered that is NP-complete and whose complement is also NP-complete. So this alone would be surprising. But, in addition, if BIN-OVERSTUFFED were NP-complete, we would have a proof that NP = co-NP. That result would also be surprising. One reason is that, although we don't know whether NP = co-NP implies P = NP, we do know that NP ≠ co-NP implies P ≠ NP. Since it is widely assumed that P ≠ NP, it is generally also assumed that NP ≠ co-NP.

23) Let G be an undirected, weighted graph with vertices V, edges E, and a function $cost(e)$ that assigns a positive cost to each edge e in E. A ***cut*** of G is a subset S of the vertices in V. The cut divides the vertices in V into two subsets, S and $V - S$. Define the ***size*** of a cut to be the sum of the costs of all edges (u, v) such that one of u or v is in S and the other is not. We'll say that a cut is ***nontrivial*** iff it is neither $\varnothing$ nor V. Recall that we saw, in Section 28.7.4, that finding shortest paths is easy (i.e., it can be done in polynomial time), but that finding longest paths is not. We'll observe a similar phenomenon with respect to cuts.

a) Sometimes we want to find the smallest cut in a graph. For example, it is possible to prove that the maximum flow between two nodes s and t is equal to the weight of the smallest cut that includes s but not t. Show that the following language is in P:

- MIN-CUT = $\{\langle G, k\rangle$: there exists a nontrivial cut of G with size at most $k\}$.

b) Sometimes we want to find the largest cut in a graph. Show that the following language is NP-complete.

- MAX-CUT = $\{\langle G, k\rangle$: there exists a cut of G with size at least $k\}$. Show that MAX-CUT is NP-complete.

c) Sometimes, when we restrict the form of a problem we wish to consider, the problem becomes easier. So we might restrict the maximum-cut problem to graphs where all edge costs are 1. It turns out that, in this case, the "simpler" problem remains NP-complete. Show that the following language is NP-complete:

- SIMPLE-MAX-CUT = $\{\langle G, k\rangle$: all edge costs in G are 1 and there exists a cut of G with size at least $k\}$.

d) Define the ***bisection*** of a graph G to be a cut where S contains exactly half of the vertices in V. Show that the following language is NP-complete: (Hint: the graph G does not have to be connected.)

- MAX-BISECTION = $\{\langle G, k\rangle$: G has a bisection of size at least $k\}$.

24) Show that each of the following functions is time-constructible:
a) $n \log n$.
b) $n\sqrt{n}$.
c) n^3.
d) 2^n.
e) $n!$.

25) In the proof of Theorem 28.27 (the Deterministic Time Hierarchy Theorem), we had to construct a string w of the form $\langle M_{t(n)easy}\rangle 10^p$. Let n be $|\langle M_{t(n)easy}\rangle 10^p|$. One of the constraints on our choice of p was that it be long enough that $|\langle M_{t(n)easy}\rangle| < \log(t(n)/\log t(n))$. Let m be $|\langle M_{t(n)easy}\rangle|$. Then we claimed that the condition would be satisfied if p is at least 2^{2^m}. Prove this claim.

26) Prove or disprove each of the following claims:

a) If $A \leq_M B$ and $B \in$ P, then $A \in$ P.

False. The mapping may not be polynomial.

b) If $A \leq_P B$ and B and C are in NP, then $A \cup C \in$ NP.

True. If $A \leq_P B$ and B is in NP, then A is in NP. The class NP is closed under complement. So $A \cup C \in$ NP.

c) Let $ndtime(f(n))$ be the set of languages that can be decided by some nondeterministic Turing machine in time $\mathcal{O}(f(n))$. Every language in $ndtime(2^n)$ is decidable.

True. If L can be decided in $ndtime(2^n)$, then, trivially, it can be decided.

d) Define a language to be ***co-finite*** iff its complement is finite. Any co-finite language is in NP.

True. Let L be any co-finite language. Since $\neg L$ is finite, it is regular. The regular languages are closed under complement, so L is also regular. By Theorem 28.2, every regular language is in P. Since P $\subseteq$ NP, L is in NP.

e) Given an alphabet Σ, let A and B be nonempty proper subsets of Σ^*. If both A and B are in NP then $A \leq_M B$.

True. Since A is in NP, it is decidable by some TM M. So the following algorithm R reduces A to B:
On input w:

1. Run M on w.
2. If M accepts, then output some element of B. It doesn't matter which one. (There must be at least one because B is nonempty.)
3. If M does not accept, then output some element of Σ^* that is not in B. Again, it doesn't matter which one. (There must be one because B is a proper subset of Σ^*).

Then $w \in A$ iff $R(w) \in B$.

f) Define the language:

- MANY-CLAUSE-SAT = $\{w : w$ is a Boolean wff in conjunctive normal form, w has m variables and k clauses, and $k \geq 2^m\}$.

If P $\neq$ NP, MANY-CLAUSE-SAT $\in$ P.

True. We can check to see whether w is satisfiable by enumerating its truth table. That takes time that is $\mathcal{O}(2^m)$. If $w \in$ MANY-CLAUSE-SAT, w has at least 2^m clauses, so its length is at least 2^m. So it is possible to check for satisfiability in time that is $\mathcal{O}(|w|)$.

29 Space Complexity Classes

1) In Section 29.1.2, we defined *MaxConfigs*(M) to be $|K|\cdot|\Gamma|^{spacereq(M)}\cdot spacereq(M)$. We then claimed that, if c is a constant greater than $|\Gamma|$, then $MaxConfigs(M) \in \mathcal{O}(c^{spacereq(M)})$. Prove this claim by proving the following more general claim:

Given: f is a function from the natural numbers to the positive reals,
f is monotonically increasing and unbounded,
a and c are positive reals, and
$1 < a < c$

Then: $f(n)\cdot a^{f(n)} \in \mathcal{O}(c^{f(n)})$.

Since $\log\frac{c}{a} > 0$, $\lim\frac{\log x}{x} = 0$ and $\frac{\log x}{x}$ is monotonically decreasing, there exists an N such that, if $x \geq N$, then $\frac{\log x}{x} \leq \log\frac{c}{a}$. Since f is monotonically increasing and is unbounded, there is a k such that, if $n \geq k, f(n) \geq N$.

Thus:
$$\frac{\log f(n)}{f(n)} \leq \log\frac{c}{a}$$

So:
$$\log f(n) \leq f(n)\cdot\log\frac{c}{a}$$
$$\leq f(n)\cdot\log c - f(n)\cdot\log a$$

So :
$$\log f(n) + f(n)\cdot\log a \leq f(n)\cdot\log c$$

And:
$$\forall n > k\,(f(n)\cdot a^{f(n)} \leq c^{f(n)})$$

Thus:
$$f(n)\cdot a^{f(n)} \in \mathcal{O}(c^{f(n)})$$

2) Prove that PSPACE is closed under:

a) Complement.

If L is in PSPACE, then it is decided by some deterministic Turing machine M with the property that *spacereq*(M) is a polynomial. Then the Turing machine M* that is identical to M except that its y and n states are reversed is a polynomial-space Turing machine that decides $\neg L$. Thus $\neg L$ is also in PSPACE.

b) Union.

If L_1 and L_2 are in PSPACE, then they are decided, in polynomial space, by some deterministic Turing machines M_1 and M_2, respectively. Then the Turing machine M* that nondeterministically chooses to run M_1 or M_2 is a polynomial-space nondeterministic Turing machine that decides $L_1 \cup L_2$. So $L_1 \cup L_2$ is in NPSPACE. By Theorem 29.3, therefore, it is also in PSPACE.

c) Concatenation.

If L_1 and L_2 are in PSPACE, then they are decided, in polynomial space, by some deterministic Turing machines M_1 and M_2, respectively. Then the following nondeterministic, two-tape Turing machine M* decides $L_1 \cup L_2$: On input w, M* nondeterministically chooses a place to divide w into two pieces. It puts the second piece on tape 2, leaving the first piece on tape 1. Then it runs M_1 on the first piece and M_2 on the second piece. If they both accept, then it accepts. Otherwise, it rejects. M* is a polynomial-space

nondeterministic Turing machine that decides L_1 L_2. So L_1 L_2 is in NPSPACE. By Theorem 29.3, therefore, it is also in PSPACE.

d) Kleene star.

If L is in PSPACE, then it is decided, in polynomial space, by some deterministic Turing machine M. Then the following nondeterministic, two-tape Turing machine M^* decides L^*: On input w, M^* begins by nondeterministically choosing a first substring to check for membership in L. It moves its choice to its second tape and runs M. If it accepts, then it nondeterministically chooses a second substring and does the same thing with it, reusing the same space on tape 2. It continues until it has consumed all of w. If it does that and if all substrings were accepted, it accepts. Otherwise, it it gets stuck, it rejects. M^* is a polynomial-space nondeterministic Turing machine that decides L^*. So L^* is in NPSPACE. By Theorem 29.3, therefore, it is also in PSPACE.

3) Define the language:

- U = {<M, w, 1^s> : M is a Turing machine that accepts w within space s}.

Prove that U is PSPACE-complete

4) In Section 28.7.3, we defined the language 2-SAT = {<w> : w is a wff in Boolean logic, w is in 2-conjunctive normal form and w is satisfiable} and saw that it is in P. Show that 2-SAT is NL-complete.

5) Prove that $A^nB^n = \{a^nb^n : n \geq 0\}$ is in L.

A^nB^n can be decided by a Turing machine M that uses its working tape to store a count, in binary, of the number of a's it sees. It then decrements the count by 1 for each b and accepts iff the count becomes 0 when the last b is read. Given an input of length k, the value of the counter is at most k. So the space required to store it in binary is $\mathcal{O}(\log k)$.

6) In Example 21.5, we described the game of Nim. We also showed an efficient technique for deciding whether or not the current player has a guaranteed win. Define the language:

- NIM = {<b>: b is a Nim configuration (i.e., a set of piles of sticks) and there is a guaranteed win for the current player}.

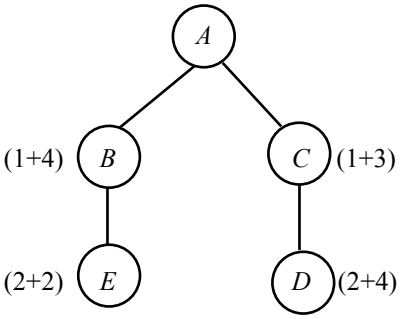
Prove that NIM $\in$ L.

7) Prove Theorem 29.12 (The Deterministic Space Hierarchy Theorem).

30 Practical Solutions for Hard Problems

1) In Exercise 28.23 we defined a cut in a graph, the size of a cut and a bisection. Let G be a graph with $2v$ vertices and m edges. Describe a randomized, polynomial-time algorithm that, on input G, outputs a cut of G with expected size at least $mv/(2v\text{-}1)$. (Hint: analyze the algorithm that takes a random bisection as its cut.)

2) Suppose that the A^* algorithm has generated the following tree so far:



Assume that the nodes were generated in the order, A, B, C, D, E. The expression (g, h) associated with each node gives the values of the functions g and h at that node.

a) What node will be expanded at the next step?

E

b) Can it be guaranteed that A^*, using the heuristic function h that it is using, will find an optimal solution? Why or why not?

We could make this guarantee if we could be sure that h never overestimates the true cost h^* of getting from a node to a goal. Since we have incomplete information about h, we cannot make that guarantee. Additionally, we notice that if $h(E)$ is right, then h overestimate $h^*(B)$.

3) Simple puzzles offer a way to explore the behavior of search algorithms such as A^*, as well as to experiment with a variety of heuristic functions. Pick one (for example the 15-puzzle of Example 4.8 or see 💻) and use A^* to solve it. Can you find an admissible heuristic function that is effective at pruning the search space?

31 Summary and References

Appendix A: Review of Mathematical Background: Logic, Sets, Relations, Functions, and Proof Techniques

1) Prove each of the following:
   a) $((A \wedge B) \rightarrow C) \leftrightarrow (\neg A \vee \neg B \vee C)$.

$((A \wedge B) \rightarrow C) \leftrightarrow (\neg A \vee \neg B \vee C) \equiv$	
$(\neg(A \wedge B) \vee C) \leftrightarrow (\neg A \vee \neg B \vee C)$	Definition of $\rightarrow$
$((\neg A \vee \neg B) \vee C) \leftrightarrow (\neg A \vee \neg B \vee C)$	de Morgan's Law
$(\neg A \vee \neg B \vee C) \leftrightarrow (\neg A \vee \neg B \vee C)$	Associativity of $\vee$
True	Definition of $\leftrightarrow$

   b) $(A \wedge \neg B \wedge \neg C) \rightarrow (A \vee \neg(B \wedge C))$.

$(A \wedge \neg B \wedge \neg C) \rightarrow (A \vee \neg(B \wedge C)) \equiv$	
$\neg(A \wedge \neg B \wedge \neg C) \vee (A \vee \neg(B \wedge C))$	Definition of $\rightarrow$
$\neg A \vee B \vee C \vee (A \vee \neg(B \wedge C))$	de Morgan's Law
$\neg A \vee B \vee C \vee A \vee \neg(B \wedge C)$	Associativity of $\vee$
$\neg A \vee B \vee C \vee A \vee \neg B \vee \neg C$	de Morgan's Law
$\neg A \vee A \vee B \neg B \vee C \vee \neg C$	Commutativity of $\vee$
$(\neg A \vee A) \vee (B \neg B) \vee (C \vee \neg C)$	Associativity of $\vee$
True $\vee$ *True* $\vee$ *True*	Definition of $\vee$
True	Definition of $\vee$

2) List the elements of each of the following sets:
   a) $\mathcal{P}(\{\text{apple, pear, banana}\})$.

$\varnothing$, {apple}, {pear}, {banana}, {apple, pear}, {apple, banana}, {pear, banana}, {apple, pear,banana}

   b) $\mathcal{P}(\{a, b\}) - \mathcal{P}(\{a, c\})$.

{b}, {a, b}

   c) $\mathcal{P}(\varnothing)$.

$\varnothing$

   d) $\{a, b\} \times \{1, 2, 3\} \times \varnothing$.

$\varnothing$

   e) $\{x \in \mathbb{N}: (x \leq 7 \wedge x \geq 7\}$.

7

   f) $\{x \in \mathbb{N}: \exists y \in \mathbb{N} (y < 10 \wedge (y + 2 = x))\}$ (where $\mathbb{N}$ is the set of nonnegative integers).

2, 3, 4, 5, 6, 7, 8, 9, 10, 11

   g) $\{x \in \mathbb{N}: \exists y \in \mathbb{N} (\exists z \in \mathbb{N} ((x = y + z) \wedge (y < 5) \wedge (z < 4)))\}$.

0, 1, 2, 3, 4, 5, 6, 7

3) Prove each of the following:
   a) $A \cup (B \cap C \cap D) = (A \cup B) \cap (A \cup D) \cap (A \cup C)$.

$A \cup (B \cap C \cap D) = (A \cup B) \cap (A \cup D) \cap (A \cup C)$. Set union distributes over set intersection.

   b) $A \cup (B \cap C \cap A) = A$.
   c) $(B \cap C) - A \subseteq C$.

4) Consider the English sentence, "If some bakery sells stale bread and some hotel sells flat soda, then the only thing everyone likes is tea." This sentence has at least two meanings. Write two (logically different) first-order-logic sentences that correspond to meanings that could be assigned to this sentence. Use the following predicates: $P(x)$ is *True* iff x is a person; $B(x)$ is *True* iff x is a bakery; $S_B(x)$ is *True* iff x sells stale bread; $H(x)$ is *True* iff x is a hotel; $S_S(x)$ is *True* iff x sells flat soda; $L(x, y)$ is *True* iff x likes y; and $T(x)$ is *True* iff x is tea.

The first meaning says that, if the condition is true, then the only thing that is liked by everyone is tea:

$$(\exists x, y\ (B(x) \wedge S_B(x) \wedge H(y) \wedge S_S(y)) \rightarrow \forall z\ (\neg T(z) \rightarrow \exists p\ (P(p) \wedge \neg L(p, z))))$$

The second meaning says that, if the condition is true, then, for each individual person, the only thing that individual likes is tea:

$$(\exists x, y\ (B(x) \wedge S_B(x) \wedge H(y) \wedge S_S(y)) \rightarrow \forall p, z\ (P(p) \wedge \neg T(z) \rightarrow \neg L(p, z)).$$

5) Let P be the set of positive integers. Let L = {A, B, …, Z} (i.e., the set of upper case characters in the English alphabet). Let T be the set of strings of one or more upper case English characters. Define the following predicates over those sets:

- For $x \in L$, $V(x)$ is *True* iff x is a vowel. (The vowels are A, E, I, O, and U.)
- For $x \in L$ and $n \in P$, $S(x, n)$ is *True* iff x can be written in n strokes.
- For $x \in L$ and $s \in T$, $O(x, s)$ is *True* iff x occurs in the string s.
- For $x, y \in L$, $B(x, y)$ is *True* iff x occurs before y in the English alphabet.
- For $x, y \in L$, $E(x, y)$ is *True* iff $x = y$.

Using these predicates, write each of the following statements as a sentence in first-order logic:
   a) A is the only upper case English character that is a vowel and that can be written in three strokes but does not occur in the string STUPID.

$\forall x \in L\ (E(x, \mathtt{A}) \leftrightarrow (V(x) \wedge S(x, 3) \wedge \neg O(x, \mathtt{STUPID})))$.

   b) There is an upper case English character strictly between K and R that can be written in one stroke.

$(\exists x \in L\ (B(\mathtt{K}, x) \wedge B(x, \mathtt{R}) \wedge S(x, 1))$.

6) Choose a set A and predicate P and then express the set {1, 4, 9, 16, 25, 36, …} in the form:

$\{x \in A : P(x)\}$.

Let A = $\mathbb{N}$. Then write $\{x \in \mathbb{N} : x \neq 0$ and $\sqrt{x} \in \mathbb{N}\}$.

7) Find a set that has a subset but no proper subset.

The only such set is $\varnothing$.

8) Give an example, other than one of the ones in the book, of a relation on the set of people that is reflexive and symmetric but not transitive.

Lives-within-a-mile-of

9) Not equal (defined on the integers) is (circle all that apply): reflexive, symmetric, transitive.

Symmetric only.

10) In Section A.3.3, we showed a table that listed the eight possible combinations of the three properties: reflexive, symmetric and transitive. Add antisymmetry to the table. There are now 16 possible combinations. Which combinations could some nontrivial binary relation posses? Justify your answer with examples to show the combinations that are possible and proofs of the impossibility of the others.

Somewhere in one of Alan's handouts. 14 are possible.

11) Using the definition of $\equiv_p$ (equivalence modulo p) that is given in Example A.4, let R_p be a binary relation on $\mathbb{N}$, defined as follows, for any $p \geq 1$:

$R_p = \{(a, b): a \equiv_p b\}$

So, for example, R_3 contains (0, 0), (0, 3), (6, 9), (1, 4), etc., but does not contain (0, 1), (3, 4), etc.

a) Is R_p an equivalence relation for every $p \geq 1$? Prove your answer.

Yes, R_p is an equivalence relation for every $p \geq 1$:

- It is reflexive since, for all $a \in \mathbb{N}$, $a \equiv_p a$.
- It is symmetric since, for all $a, b \in \mathbb{N}$, $a \equiv_p b \rightarrow b \equiv_p a$.
- It is transitive since, for all $a, b, c \in \mathbb{N}$, $a \equiv_p b \wedge b \equiv_p c \rightarrow a \equiv_p c$.

b) If R_p is an equivalence relation, how many equivalence classes does R_p induce for a given value of p? What are they? (Any concise description is fine.)

There are p equivalence classes. The i^{th} equivalence class contains those elements that are equal to i modulo p.

c) Is R_p a partial order? A total order? Prove your answer.

Neither. For R_p to be a partial order, it would have to be antisymmetric. Since it's symmetric, it can't be antisymmetric. Since every total order is also a partial order, R_p is also not total.

12) Let $S = \{w \in \{a, b\}^*\}$. Define the relation *Substr* on the set S to be $\{(s, t) : s \text{ is a substring of } t\}$.

a) Choose a small subset of *Substr* and draw it as a graph (in the same way that we drew the graph of Example A.5.

b) Is *Substr* a partial order?

13) Let P be the set of people. Define the function:

father-of: $P \rightarrow P$.
father-of(x) = the person who is x's father

a) Is *father-of* one-to-one?

No. *father-of*(*John Quincy Adams*) = *John Adams* and *father-of*(*Charles Adams*) = *John Adams*.

b) Is it onto?

No. There is no element x of P such that *father-of*(x) = *Abigail Adams*.

14) Are the following sets closed under the following operations? If not, give an example that proves that they are not and then specify what the closure is.

a) The negative integers under subtraction.

The negative integers are not closed under subtraction. For example (-4) – (-6) = 2. So the closure is the set of integers.

b) The negative integers under division.

The negative integers are not closed under division. For example, -4/-2 = 2, so the positive integers must be added to the closure. But 2/4 is not an integer. So the closure is the set of rational numbers – {0}.

c) The positive integers under exponentiation.

The positive integers are closed under exponentiation.

d) The finite sets under Cartesian product.

The finite sets are closed under cross product. Given two finite sets, x and y, $|x \times y| = |x| * |y|$.

e) The odd integers under remainder, mod 3.

The odd integers are not closed under remainder, mod 3. The range of this function is {0, 1, 2}. So the closure is the odd integers union with {0, 2}.

f) The rational numbers under addition.

By construction. If x and y are rational, then they can be represented as a/b and c/d, where a, b, c, and d are integers. The sum of x and y is then:

$$\frac{ad + cb}{bd}$$

Since the integers are closed under both addition and multiplication, both the numerator and the denominator are integers. Since neither b nor d is 0, neither is the denominator. So the result is rational.

15) Give examples to show that:

a) The intersection of two countably infinite sets can be finite.

Let x be a natural number: $\{x \geq 0\} \cap \{x \leq 0\} = \{0\}$.

b) The intersection of two countably infinite sets can be countably infinte.

Let x be a natural number: $\{x \geq 0\} \cap \{x \geq 0\} = \{x \geq 0\}$.

c) The intersection of two uncountable sets can be finite.

Let x be a real number: $\{x \geq 0\} \cap \{x \leq 0\} = \{0\}$.

d) The intersection of two uncountable sets can be countably infinite.

Let S_1 be the set of all positive real numbers. Let S_2 be the set of all negative real numbers union the set of all positive integers. Both S_1 and S_2 are uncountably infinite. But $S_1 \cap S_2$ = the set of all positive integers, which is countably infinite.

e) The intersection of two uncountable sets can be uncountable.

Let S be the set of real numbers. $S \cap S = S$.

16) Let R = {(1, 2), (2, 3), (3, 5), (5, 7), (7, 11), (11, 13), (4, 6), (6, 8), (8, 9), (9, 10), (10, 12)}. Draw a directed graph representing R^*, the reflexive, transitive closure of R.

I'll write this out instead of drawing it. Let A be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}. Then we have:
{(1, 1), (2, 2),(3, 3)….,
(1, 2), (1, 3), (1, 5), (1, 7), … where 1 is related to all primes in A
(3, 5), (3, 7), (3, 11), where 3 is related to all primes in A that are less greater than it
(5, 7), as for 3
(4, 6), similarly for all non primes, which are related to all other nonprimesin A and less then themselves}

17) Let $\mathbb{N}$ be the set of nonnegative integers. For each of the following sentences in first-order logic, state whether the sentence is valid, is not valid but is satisfiable, or is unsatisfiable. Assume the standard interpretation for < and >. Assume that f could be any function on the integers. Prove your answer.

a) $\forall x \in \mathbb{N}\ (\exists y \in \mathbb{N}\ (y < x))$.

Unsatisfiable. The proof is by counterexample. If $x = 0$, there is no natural number y that is less than x.

b) $\forall x \in \mathbb{N}\ (\exists y \in \mathbb{N}\ (y > x))$.

Tautology. The proof is by construction. For any number x, one such y is $x + 1$.

c) $\forall x \in \mathbb{N}\ (\exists y \in \mathbb{N}\ f(x) = y)$.

Satisfiable but not a tautology. Let f be the identity function. Then the sentence is true. Let f be the function divide by 2. Then, for any odd value of x, there is no natural number y such that $f(x) = y$.

18) Let $\mathbb{N}$ be the set of nonnegative integers. Let A be the set of nonnegative integers x such that $x \equiv_3 0$. Show that $|\mathbb{N}| = |A|$.

We show a bijection f that maps from A to $\mathbb{N}$: $f(x) = x/3$.

19) What is the cardinality of each of the following sets? Prove your answer

a) $\{n \in \mathbb{N} : n \equiv_3 0\}$.

(Same as for problem 18) We show a bijection f that maps from $\{n \in \mathbb{N} : n \equiv_3 0\}$ to $\mathbb{N}$: $f(x) = x/3$.

b) $\{n \in \mathbb{N} : n \equiv_3 0\} \cap \{n \in \mathbb{N} : n \text{ is prime}\}$.

The only prime number that is evenly divisible by 3 is 3. So the cardinality of this set is 1.

c) $\{n \in \mathbb{N} : n \equiv_3 0\} \cup \{n \in \mathbb{N} : n \text{ is prime}\}$.

Both $\{n \in \mathbb{N} : n \equiv_3 0\}$ and $\{n \in \mathbb{N} : n \text{ is prime}\}$ are countably infinite. The following algorithm constructs an infinite enumeration of their union U:

Consider the elements of $\mathbb{N}$ in order. For each do:
See if it is evenly divisible by 3.
See if it is prime.
If it passes both tests, enumerate it.

We thus have an infinite enumerate of U. By Theorem A.1, a set is countably infinite iff there exists an infinite enumeration of it. So U iscountably infinite.

20) Prove that the set of rational numbers is countably infinite.

21) Use induction to prove each of the following claims:

a) $\forall n>0 \left(\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}\right)$.

b) $\forall n>0 \ (n! \geq 2^{n-1})$. Recall that $0! = 1$ and $\forall n > 0 \ (n! = n(n-1)(n-2) \ldots 1)$.

Base step: If $n = 1$, then $1! = 1 = 2^0$

So $1! \geq 2^0$

Prove that $(n! \geq 2^{n-1}) \rightarrow ((n+1)! \geq 2^{n+1-1})$:

$(n+1)!$	$= (n+1)n!$	definition of factorial
	$\geq (n+1)2^{n-1}$	induction hypothesis
	$\geq (2)2^{n-1}$	since $(n+1) \geq 2$
	$\geq 2^n$	
	$\geq 2^{n+1-1}$	

c) $\forall n>0 \left(\sum_{k=0}^{n} 2^k = 2^{n+1}-1\right)$.

Base step: If $n = 0$, then we have $\sum_{k=0}^{0} 2^k = 2^0 = 1 = 2^{0+1} - 1)$.

Prove that $\left(\sum_{k=0}^{n} 2^k = 2^{n+1}-1\right) \rightarrow \left(\sum_{k=0}^{n+1} 2^k = 2^{n+1+1}-1\right)$:

$$\sum_{k=0}^{n+1} 2^k = \sum_{k=0}^{n} 2^k + 2^{n+1} = 2^{n+1}-1 + 2^{n+1} = 2 \cdot 2^{n+1}-1 = 2^{(n+1)+1}-1.$$

d) $\forall n \geq 0 \left(\sum_{k=0}^{n} r^k = \frac{r^{n+1}-1}{r-1}\right)$, given $r \neq 0,1$.

Base step: If $n = 0$, then we have $\sum_{k=0}^{0} r^k = 1 = \frac{r^{0+1}-1}{r-1}$.

Prove that $(\forall r \neq 0,1\ (\forall n \geq 0\ (\sum_{k=0}^{n} r^k = \frac{r^{n+1}-1}{r-1}))) \rightarrow (\forall r \neq 0,1\ (\sum_{k=0}^{n+1} r^k = \frac{r^{(n+1)+1}-1}{r-1}))$:

$$\begin{aligned}
\sum_{k=0}^{n+1} r^k &= \sum_{k=0}^{n} r^k + r^{n+1} \\
&= \frac{r^{n+1}-1}{r-1} + r^{n+1} \\
&= \frac{r^{n+1}-1}{r-1} + \frac{r^{n+2}-r^{n+1}}{r-1} \\
&= \frac{r^{(n+1)+1}-1}{r-1}
\end{aligned}$$

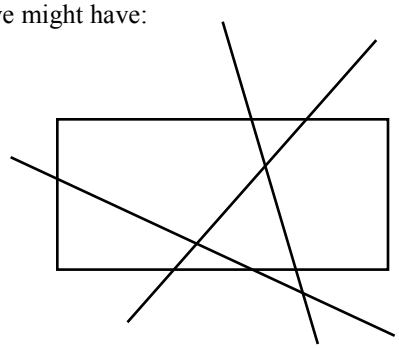
e) $\forall n \geq 0 \left(\sum_{k=0}^{n} f_k^2 = f_n \cdot f_{n+1}\right)$, where f_n is the n^{th} element of the Fibonacci sequence, as defined in Example 24.4.

Base step: If $n = 0$, then we have $\sum_{k=0}^{0} f_k^2 = f_0 \cdot f_1 = 1$.

Prove that $(\forall n \geq 0\ (\sum_{k=0}^{n} f_k^2 = f_n \cdot f_{n+1})) \rightarrow (\sum_{k=0}^{n+1} f_k^2 = f_{n+1} \cdot f_{(n+1)+1})$:

$$\begin{aligned}
\sum_{k=0}^{n+1} f_k^2 &= \sum_{k=0}^{n} f_k^2 + f_{n+1}^2 \\
&= f_n \cdot f_{n+1} + f_{n+1}^2 \\
&= f_{n+1} \cdot (f_n \cdot + f_{n+1}) \\
&= f_{n+1} \cdot f_{(n+1)+1}
\end{aligned}$$

22) Consider a finite rectangle in the plane. We will draw (infinite) lines that cut through the rectangle. So, for example, we might have:



In Section 28.7.6, we define what we mean when we say that a map can be colored using two colors. Treat the rectangle that we just drew as a map, with regions defined by the lines that cut through it. Use induction to prove that, no matter how many lines we draw, the rectangle can be colored using two colors.

Let $P(n)$ = Any map with n lines cutting it can be colored (with no two adjacent regions being the same color) with two colors. Call them red and black.

Base case: Let $n = 1$. The map has two regions. Color one red and one black.

Prove that $P(n) \rightarrow P(n+1)$. Given a map with $n+1$ lines cutting it, choose one line and remove it. The resulting map is cut by n lines. By the induction hypothesis, this map can be colored with two colors. Now add back the $n+1^{st}$ line that was removed and, on one side of it, reverse all the colors. Consider every region in this new map and consider every line segment S that forms a boundary of that region. Either:

- S lies on the reintroduced $n+1^{st}$ line: In this case, S cuts across what was a single region before the $n+1^{st}$ line was reintroduced. The region previously had only a single color and we swapped that color on one side of the new line and not the other. So the regions on the two sides of S must be different colors.
- S lies on some other line. In this case, it's a line that was there before the $n+1^{st}$ was reintroduced. The regions on the two sides of it thus lie on the same side of the $n+1^{st}$ line. That means that either they both stayed the same color or they both got reversed. In either case, since, by the induction hypothesis, they were different colors before the $n+1^{st}$ line was added, they still are.

No line segment S can satisfy both of these conditions. So we have that, in all cases, adjacent regions must be of different colors.

23) Let $div_2(n) = \lfloor n/2 \rfloor$ (i.e., the largest integer that is less than or equal to $n/2$). Alternatively, think of it as the function that performs division by 2 on a binary number by shifting right one digit. Prove that the following program correctly multiplies two natural numbers. Clearly state the loop invariant that you are using.

```
mult(n, m: natural numbers) =
        result = 0.
        While m ≠ 0 do
                If odd(m) then result = result + n.
                n = 2n.
                m = div2(m).
```

24) Prove that the following program computes the function *double*(*s*) where, for any string *s*, *double*(*s*) = *True* if *s* contains at least one pair of adjacent characters that are identical and *False* otherwise. Clearly state the loop invariant that you are using.

```
double(s: string) =
        found = False.
        for i = 1 to length(s) - 1 do
                if s[i] = s[i + 1] then found = True.
        return(found).
```

We use the invariant $I \equiv$ If *s* contains a double character starting anywhere before position *i* then *found* = *True*, else *found* = *False*. On exit from the loop, we have:

$I \wedge i = length(s)$

Appendix B: The Theory: Working with Logical Formulas

1) Convert each of the following Boolean formulas to conjunctive normal form:
   a) $(a \wedge b) \rightarrow c$.

$(a \wedge b) \rightarrow c$
$\neg(a \wedge b) \vee c$
$\neg a \vee \neg b \vee c$

   b) $\neg(a \rightarrow (b \wedge c))$.

$\neg(a \rightarrow (b \wedge c))$
$\neg(\neg a \vee (b \wedge c))$
$a \wedge \neg(b \wedge c)$
$a \wedge \neg b \vee \neg c)$

   c) $(a \vee b) \rightarrow (c \wedge d)$.

$(a \vee b) \rightarrow (c \wedge d)$
$\neg(a \vee b) \vee (c \wedge d)$
$(\neg a \wedge \neg b) \vee (c \wedge d)$
$(\neg a \vee c) \wedge (\neg a \vee d) \wedge (\neg b \vee c) \wedge (\neg b \wedge d)$

   d) $\neg(p \rightarrow \neg(q \vee (\neg r \wedge s)))$.

$\neg(p \rightarrow \neg(q \vee (\neg r \wedge s)))$
$\neg(\neg p \vee \neg(q \vee (\neg r \wedge s)))$
$p \wedge (q \vee (\neg r \wedge s))$
$p \wedge (q \vee \neg r) \wedge (q \vee s)$

2) For each of the following formulas w, use *3-conjunctiveBoolean* to construct a formula w' that is satisfiable iff w is:
   a) $(a \vee b) \wedge (a \wedge \neg b \wedge \neg c \wedge d \wedge e)$

$(a \vee b) \wedge (a \wedge \neg b \wedge \neg c \wedge d \wedge e)$
$(a \vee b) \wedge a \wedge \neg b \wedge \neg c \wedge d \wedge e$ /* In conjunctive normal form now.
$(a \vee b \vee b) \wedge (a \vee a \vee a) \wedge (\neg b \vee \neg b \vee \neg b) \wedge (\neg c \vee \neg c \vee \neg c) \wedge (d \vee d \vee d) \wedge (e \vee e \vee e)$

   b) $\neg(a \rightarrow (b \wedge c))$

$\neg(a \rightarrow (b \wedge c))$
$\neg(\neg a \vee (b \wedge c))$
$a \wedge \neg(b \wedge c)$
$a \wedge \neg b \vee \neg c$

3) Convert each of the following Boolean formulas to disjunctive normal form:
   a) $(a \vee b) \wedge (c \vee d)$

$(a \vee b) \wedge (c \vee d)$
$(a \wedge (c \vee d)) \vee (b \wedge (c \vee d))$
$((a \wedge c) \vee ((a \wedge d)) \vee ((b \wedge c) \vee (b \wedge d))$
$(a \wedge c) \vee ((a \wedge d) \vee (b \wedge c) \vee (b \wedge d)$

b) $(a \vee b) \rightarrow (c \wedge d)$

$(a \vee b) \rightarrow (c \wedge d)$
$\neg(a \vee b) \vee (c \wedge d)$
$(\neg a \wedge \neg b) \vee (c \wedge d)$

4) Use a truth table to show that Boolean resolution is sound.

We need to show that, whenever $(P \vee Q)$ and $(R \vee \neg Q)$ are both true, so is $(P \vee R)$. So we construct the following table:

	P	Q	R	$P \vee Q$	$R \vee \neg Q$	$(P \vee Q) \wedge (R \vee \neg Q)$	$P \vee R$
1	T	T	T	T	T	T	T
2	T	T	F	T	F	F	T
3	T	F	T	T	T	T	T
4	T	F	F	T	T	T	T
5	F	T	T	T	T	T	T
6	F	T	F	T	F	F	F
7	F	F	T	F	T	F	T
8	F	F	F	F	T	F	F

$(P \vee Q)$ and $(R \vee \neg Q)$ are both true in rows 1, 3, 4, and 5. $(P \vee R)$ is also true in those rows.

5) Use resolution to show that the following premises are inconsistent:

$a \vee \neg b \vee c$, $b \vee \neg d$, $\neg c \vee d$, $b \vee c \vee d$, $\neg a \vee \neg b$, and $\neg d \vee \neg b$.

Numbering the premises:
(1) $a \vee \neg b \vee c$
(2) $b \vee \neg d$
(3) $\neg c \vee d$
(4) $b \vee c \vee d$
(5) $\neg a \vee \neg b$
(6) $\neg d \vee \neg b$

Resolving:
(7) $\neg d$ (2) and (6)
(8) $\neg c$ (3) and (7)
(9) $b \vee c$ (4) and (7)
(10) b (8) and (9)
(11) $a \vee c$ (1) and(10)
(12) a (8) and (11)
(13) $\neg a$ (5) and (10)
(14) *nil* (12) and (13)

6) Prove that the conclusion $b \wedge c$ follows from the premises: $a \rightarrow (c \vee d)$, $b \rightarrow a$, $d \rightarrow c$, and b.

a) Convert the premises and the negation of the conclusion to conjunctive normal form.

Converting the premises to conjunctive normal form:

(1) $\neg a \vee c \vee d$
(2) $\neg b \vee a$
(3) $\neg d \vee c$
(4) b

Negating the conclusion: $\neg(b \wedge c)$. Then converting it to conjunctive normal form:

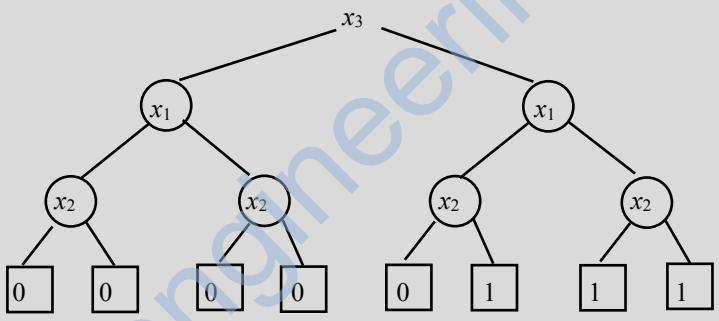
(5) $\neg b \vee \neg c$

b) Use resolution to prove the conclusion.

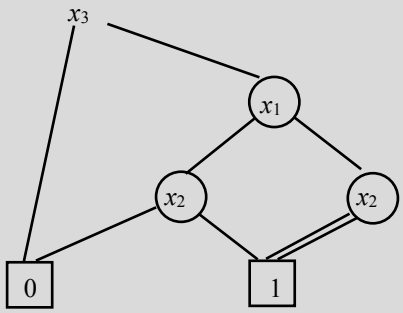
(6) $\neg c$	(4) and (5)
(7) $\neg d$	(3) and (6)
(8) $\neg a \vee c$	(1) and (7)
(9) $\neg a$	(6) and (8)
(10) $\neg b$	(2) and (9)
(11) *nil*	(4) and (10)

7) Consider the Boolean function $f_1(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge x_3$. that we used as an example in B.1.3. Show how f_1 can be converted to an OBDD using the variable ordering $(x_3 < x_1 < x_2)$.

We begin by building:



This tree is more collapsible than the one we got with the ordering $(x_1 < x_2 < x_3)$. This time, we can collapse to:



8) In this problem, we consider the importance of standardizing apart the variables that occur in a first-order sentence in clause form. Assume that we are given a single axiom, $\forall x$ (*Likes*(*x*, *Ice cream*)). And we want to

prove $\exists x\ (Likes(Mikey, x))$. Use resolution to do this but don't standardize apart the two occurrences of x. What happens?

Converting the axiom to clause form, we get: $Likes(x, Ice\ cream)$ [1]
To prove $\exists x\ (Likes(Mikey, x))$, we negate to get:
$\neg(\exists x\ (Likes(Mikey, x)))$
Converting this to clause form, we get:
$\forall x\ \neg(Likes(Mikey, x))$, and then: $\neg Likes(Mikey, x)$ [2]

Now we attempt to resolve [1] and [2]. To do that we attempt to unify:

$Likes(x, \quad Ice\ cream)$
$Likes(Mikey, \quad x \quad)$

We first unify *Mikey* with x, producing the substitution *Mikey*/x. Then we apply that to the remainders of the two clauses, producing:

$Ice\ cream)$
$Mikey$

Unification fails, and resolution cannot proceed.

9) Begin with the following fact from Example B.6:

[1] $\forall x\ ((Roman(x) \wedge Know(x, Marcus)) \rightarrow$
$(Hate(x, Caesar) \vee \forall y\ (\exists z\ (Hate(y, z)) \rightarrow Thinkcrazy(x, y))))$

Add the following facts:

[2] $\forall x\ ((Roman(x) \wedge Gladiator(x)) \rightarrow Know(x, Marcus))$
[3] $Roman(Claudius)$
[4] $\neg\exists x\ (Thinkcrazy(Claudius, x))$
[5] $\neg\exists x\ (Hate(Claudius, x))$
[6] $Hate(Isaac, Caesar)$
[7] $\forall x\ ((Roman(x) \wedge Famous(x)) \rightarrow (Politician(x) \vee Gladiator(x)))$
[8] $Famous(Isaac)$
[9] $Roman(Isaac)$
[10] $\neg Know(Isaac, Marcus)$

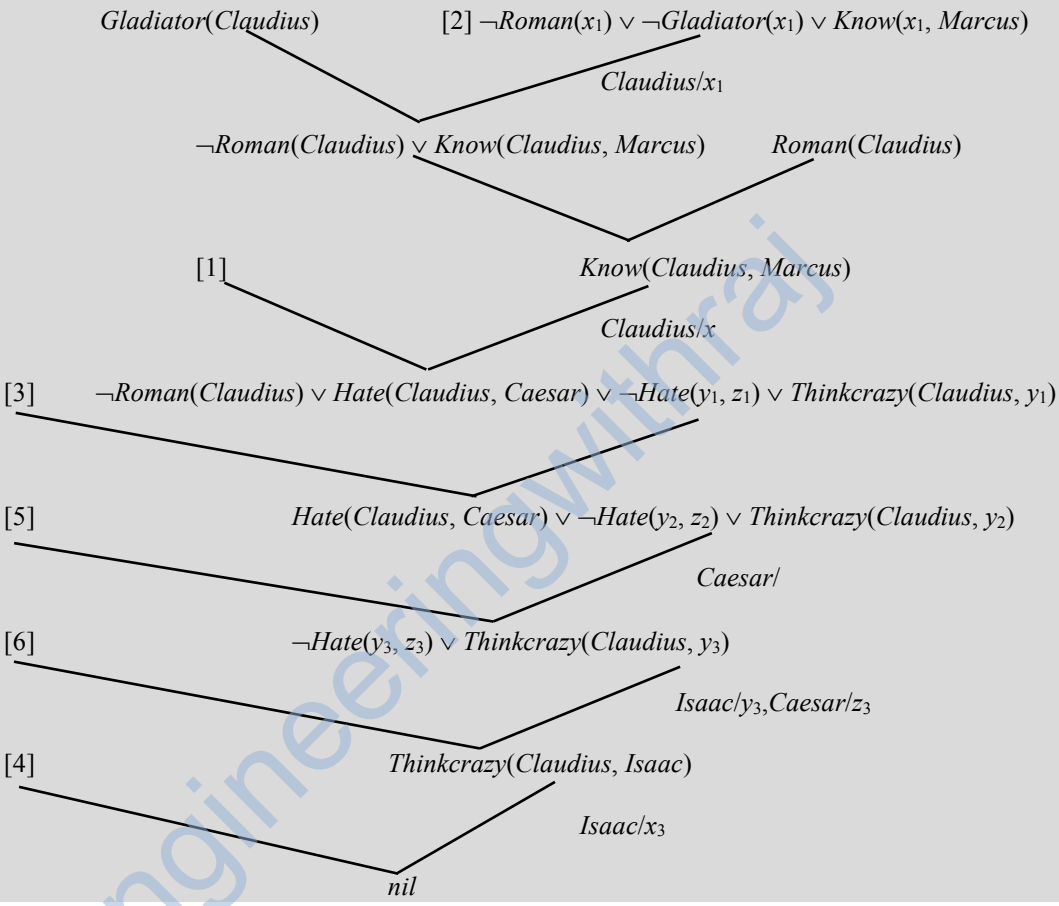
a) Convert each of these facts to clause form.

[2] $\neg Roman(x_1) \vee \neg Gladiator(x_1) \vee Know(x_1, Marcus)$
[3] $Roman(Claudius)$
[4] $\neg Thinkcrazy(Claudius, x_2)$
[5] $\neg Hate(Claudius, x_3)$
[6] $Hate(Isaac, Caesar)$
[7] $\neg Roman(x_4) \vee \neg Famous(x_4) \vee Politician(x_4) \vee Gladiator(x_4)$
[8] $Famous(Isaac)$
[9] $Roman(Isaac)$
[10] $\neg Know(Isaac, Marcus)$

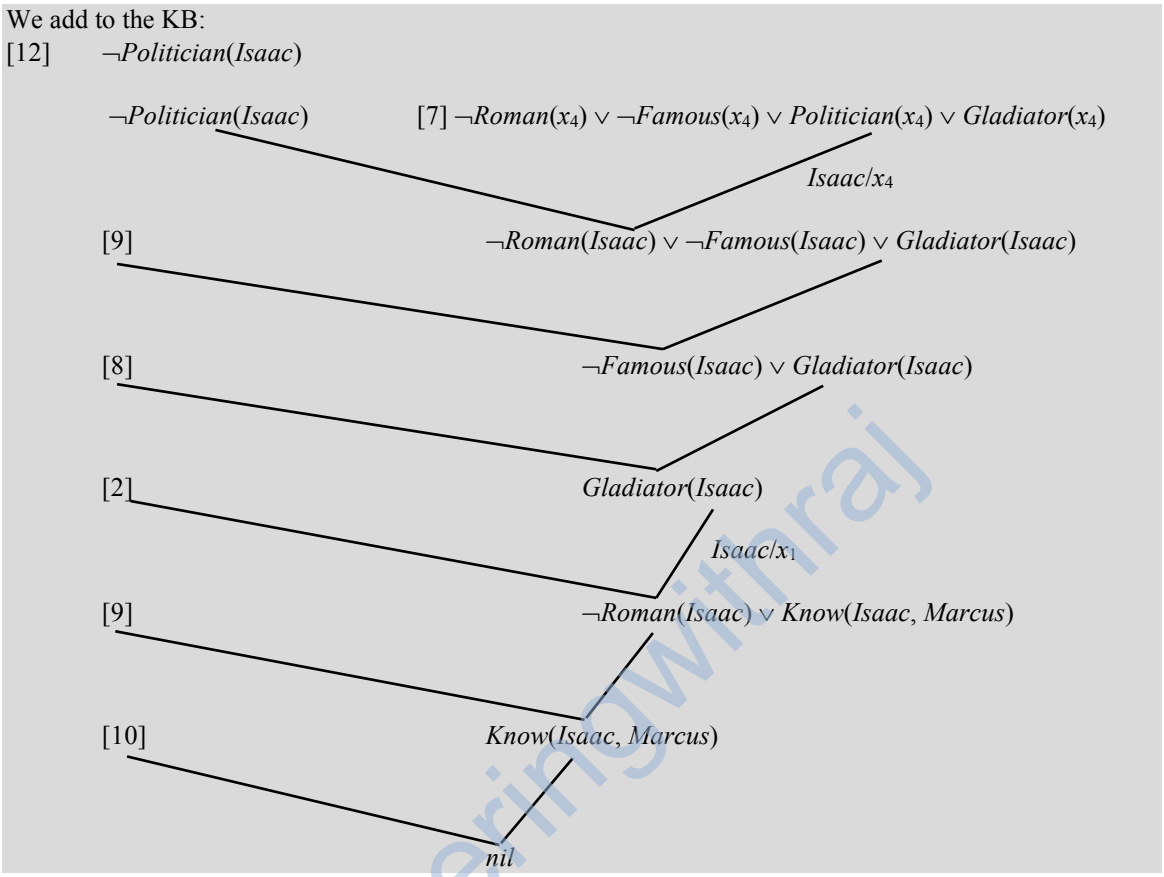
b) Use resolution and this knowledge base to prove ¬*Gladiator*(*Claudius*).

We add to the KB:
[11] *Gladiator*(*Claudius*)

And we have [1], in clause form:
[1] ¬*Roman*(*x*) ∨ ¬*Know*(*x*, *Marcus*) ∨ *Hate*(*x*, *Caesar*) ∨ ¬*Hate*(*y*, *z*) ∨ *Thinkcrazy*(*x*, *y*)



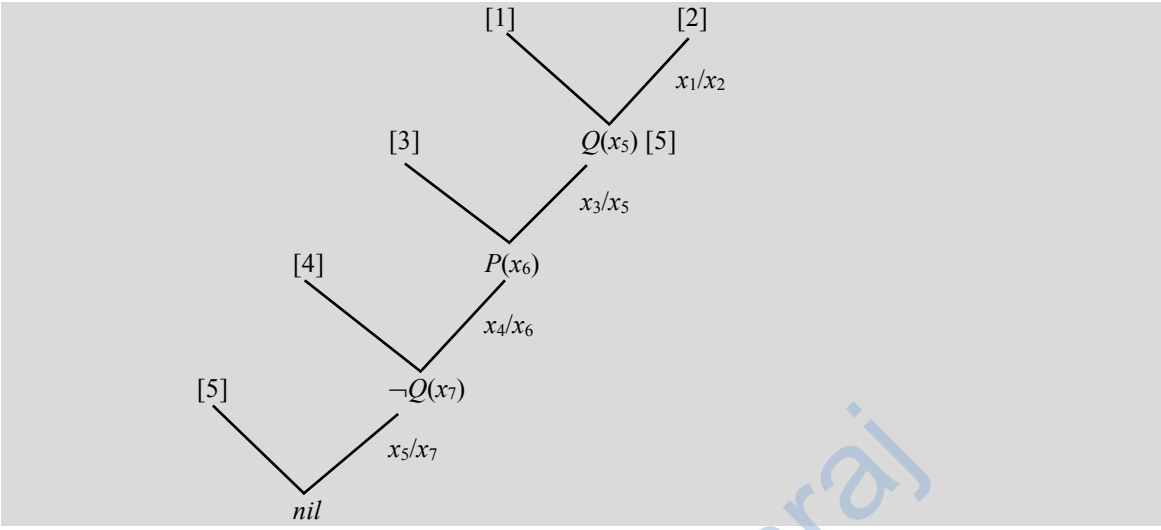
c) Use resolution and this knowledge base to prove *Politician*(*Isaac*).



10) In M.2.3, we describe a restricted form of first-order resolution called SLD resolution. This problem explores an issue that arises in that discussion. In particular, we wish to show that SLD resolution is not refutation-complete for knowledge bases that are not in Horn clause form. Consider the following knowledge base *B* (that is not in Horn clause form):

[1] $P(x_1) \vee Q(x_1)$
[2] $\neg P(x_2) \vee Q(x_2)$
[3] $P(x_3) \vee \neg Q(x_3)$
[4] $\neg P(x_4) \vee \neg Q(x_4)$

a) Use resolution to show that *B* is inconsistent (i.e., show that the empty clause *nil* can be derived).



b) Show that SLD resolution cannot derive *nil* from *B*.

The problem is that, in SLD resolution, each step must resolve with one clause that was generated by an earlier resolution step and one clause that was in the original knowledge base B. It is not allowed either to:
Resolve two clauses from the original KB, or
Resolve two clauses that were generated by a previous resolution step.

To solve this problem, it is necessary, as shown in the resolution proof given above, to resolve two clauses that arose from previous resolution steps. By exhaustively enumerating the other ways of attempting to resolve the formulas as given, we can show that no other path succeeds either.